Masaryk University
Faculty of Informatics

DeepSaber: Deep Learning for high dimensional choreography

Bachelor's Thesis

Ronald Luc

Brno, Spring 2020

Masaryk University
Faculty of Informatics

DeepSaber: Deep Learning for high dimensional choreography

Bachelor's Thesis

Ronald Luc

Brno, Spring 2020

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Ronald Luc

Advisor: doc. RNDr. Tomáš Brázdil, Ph.D.

Acknowledgements

I wish to thank Tomáš Brázdil, my supervisor at the Faculty of Informatics, Masaryk University, for his support on a project that is important to me and his guidance on writing this work.

Special thanks to Jan Pokorný for introducing me to the Beat Saber, helping me with the initial data collection and testing. It was supposed to be a weekend project.

I also wish to thank Petr Zelina for discussing new metrics at any time of the day.

Lastly, I would like to thank Demetrio Rodríguez Tereshkin and Martin Mikšík for proofreading my dyslectic text, and I hope the graphics make it an easy read for anyone else with a similar disability.

Abstract

Beat Saber is the most popular virtual reality game of 2019. The goal of the game is to slice incoming cubes in the correct direction with two virtual lightsabers. The cubes are defined by a beat map, which is synchronized with the music. The creation of a beat map takes hours.

We automate the process of beat map creation, also called learning to choreograph. The task decomposes into two sub-tasks: deciding the time of used beats (action placement) and deciding where in space to put the cubes associated with the chosen beats (action selection). We focus on action selection as the approach from the paper Dance Dance Convolution by Donahue et al. fails in the high-dimensional action space of Beat Saber.

We present a synthetic analogy dataset for Beat Saber actions and two new metrics based on word vectors. A local metric to measure the similarity between actions and a new global metric to measure the similarity between human a synthetic choreography based on the distribution of new patterns.

Using the metrics, we explore and evaluate several RNN models with a focus on solving the high-dimensionality. The final model keeps high variance and learns common patterns used by the community.

Keywords

data science, machine learning, deep learning, neural networks, RNN, word vectors, sound processing, learning to choreograph, action selection, Beat Saber

Contents

1 Introduction

Rhythm-based video games are only as good as the songs and beat maps created for them. Beat Saber is a virtual reality video game which makes beat map creation especially hard[1]. One beat map takes over 10 hours to create as the beat elements have to be placed in three dimensions.

This thesis aims to automate the process of beat map creation. The goal is to create novel local and global metrics suited for game choreography and use the metrics to evaluate different deep learning approaches.

1.1 Used terminology

Beat Saber, Guitar Hero, Beatmania, and Dance Dance Revolution are main titles from the family of rhythm-based video games. In this genre of games, the goal for the player is to press buttons, stump on dance pad or move in VR based on musical and visual cues. The cue is usually represented as a beat element approaching a beat area [Figure 1.1]. Beat map is a sequence of actions for a specific song and meta-data. Action is a set of beat elements with the same timestamp. Beat element is defined by attributes (position, rotation, duration, timestamp, etc.). When the beat area and beat element collide, the corresponding action should be performed by the player.

1.2 Beat Saber

Beat Saber[2] is the most popular virtual reality game of 2019 [1, 2]. The goal is to slice incoming beat elements (cubes with an arrow) in the correct direction with two differently colored virtual lightsabers [Figure 1.2]. The beat element attributes are x coordinate, y coordinate, rotation, hand (corresponding to red and blue colors), and timestamp since the beginning of the song in beats. The relevant beat map meta-

1. The Official Level Editor Tutorial: youtu.be/5Ex6sOEVgrM
2. Inside headset view: youtu.be/7WQtYMA8Zok

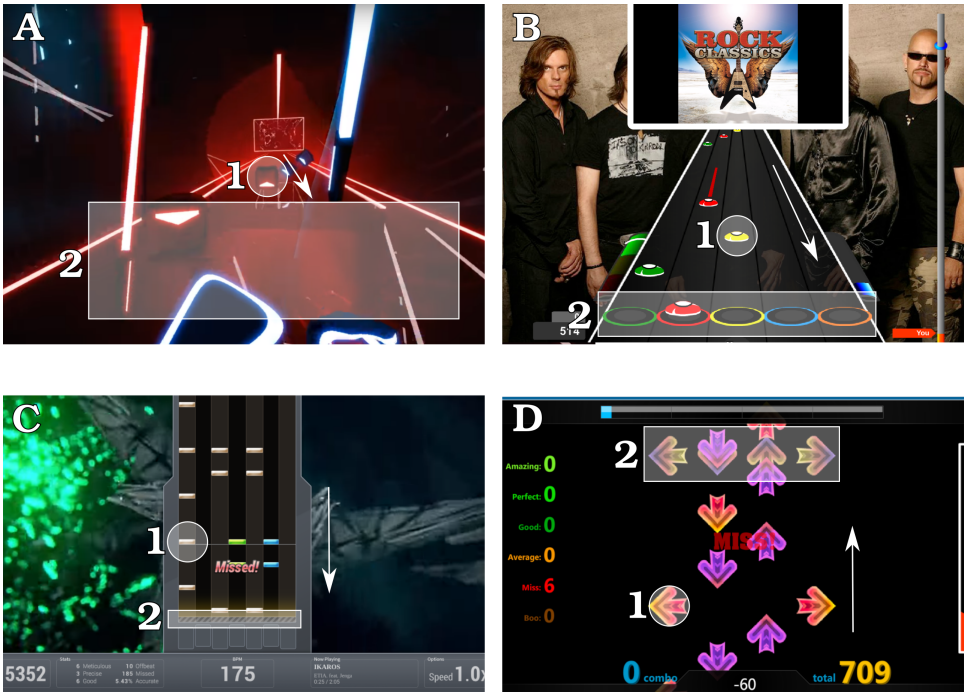

Figure 1.1: Comparison of beat elements (**1**) and beat areas (**2**) in Beat Saber (**A**), Guitar Hero (**B**), Beatmania (**C**) and Dance Dance Revolution (**D**). Arrows show movement direction of beat elements.

Figure 1.2: Slicing an incoming beat element (attributes x: 1, y: 0, rotation: down, hand: left) with red lightsaber from outside and inside VR

data are initial beats per minute value and a list of changes to beats per minute.

1.3 Background and prior art

The beat map creation for Beat Saber is harder than for Dance Dance Revolution or other games because of three-dimensional nature of VR games. All Beat Saber beat map editors are PC based. The official editor included with the game is only 2D. The majority of mapping community uses Mediocre Map Assistant 2 or BeatMapper.app[3] which

3. Beat Saber Modding Group Wiki: bsmg.wiki/mapping/#map-editing-resources

use 3D projection. Thus the creation of good beat maps requires 3D imagination.

The task of auto-generating game maps from music as defined by Donahue et al. [3] is called *Learning to choreograph*. The first approaches were rule-based [4], but current advancements in deep learning lead to a sweep of new publications [3, 5, 6].

All of the previous work decompose learning to choreograph into two sub-tasks: deciding the time of the beats used for actions (action placement) and deciding which action to use for each chosen beat (action selection). Let k be the number of actions in a beat map. Action placement is formally defined as a function $f : \text{audio} \rightarrow \mathbb{R}^k$, where for $a \in \text{audio} : f(a) = \{t : 0 \leq t \leq |a|, t \in \mathbb{R}\}$. Action selection is formally defined as a function $g : (\text{audio}, \mathbb{R}^k) \rightarrow (\text{action}, \mathbb{R})^k$, where for $a \in \text{audio}, T \in \mathbb{R}^k : g(a, T) = \bigcup_{t \in T}(\text{action}_t, t)$.

This thesis focuses only on action selection as beat placement already works well with the C-LSTM method introduced by Donahue et al. [3] and is directly applicable to Beat Saber (OxAI labs [7], Beat Sage [8]).

2 Data and search for the right data structures

The data format is an especially important question as we want to evaluate different learning approaches. Beat Saber stores songs in Ogg Vorbis audio files and beat maps for each difficulty separately in JSON files. Both audio and beat maps have to be transformed into NumPy arrays to be used by the deep learning framework [9].

2.1 Used dataset

Our dataset was collected from BeatSaver[1] using a Bootleg[2] plugin for advanced search. The beat maps with with too low or too high success rate were filtered out and the top 893 unique songs with beat maps for March 2019 were downloaded. Our dataset uses the outdated Beat Saber data format.

OxAI[3] dataset was downloaded in February 2020 to better compare the results of our projects by training on the same data. OxAI dataset consists of the curated beat maps from Beast Saber[4].

Both datasets were split into train, validation and test sets. The datasets contain overlapping data, so only one is used for training/evaluation at a time to prevent leaking train information into validation or test sets. More information can be found in Table 2.1, Figure 2.1, and data exploration notebook. Never seen actions for validation are computed as $|\text{val} \setminus \text{train}|$ and for test it equals $|\text{test} \setminus (\text{val} \cup \text{train})|$.

2.2 Actions representation

Beat Saber, in comparison to other rhythm-based games, is special because of the number of possible actions in each music beat. Attributes of beat elements in Dance Dance Revolution and Guitar Hero are position (four and five options, respectively) and pressed/released, the only attribute in Beatmania is position (eight options). At every

1. BeatSaver website: beatsaver.com
2. Bootleg-Beatsaver github: github.com/Zeekin/beatsaver-search
3. Oxford AI Society lab DeepSaber2: oxai.org/2019/09/24/deepsaber2.html
4. Beast Saber Curator Recommended list: bsaber.com/curator-recommended

Table 2.1: Dataset comparison

	Our dataset			OxAI dataset		
	train	**val**	**test**	**train**	**val**	**test**
songs [#]	705	86	78	613	80	72
beat maps [#]	1,615	182	179	1,499	197	152
total actions [#]	847,075	94,475	100,975	903,525	123,175	97,900
audio [h]	38.1	4.5	4.2	32.5	4.3	3.9
gameplay [h]	86.7	9.2	9.4	77.6	10.4	8.2
unique actions [#]	2,449	924	946	2,489	1,060	981
never seen actions [#]		96	99		105	79

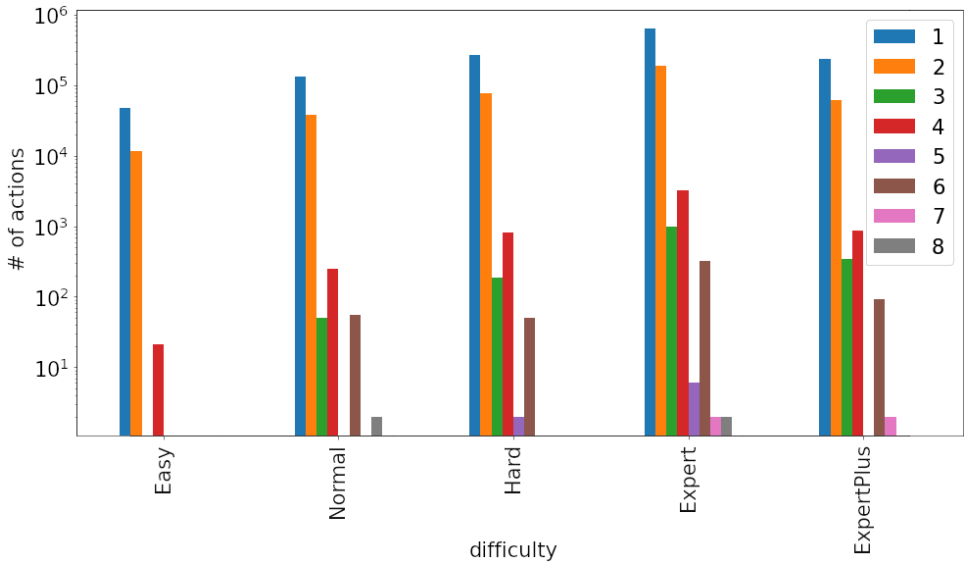

Figure 2.1: Distribution of the number of beat elements per action (**color**) per difficulty in our train dataset.

music beat, each position can be occupied by at most one beat element. Therefore Dance Dance Revolution has 81, Guitar Hero has 243, and Beatmania has 256 possible distinct actions (PDA) per beat. Possible action is a set of beat elements that can exist in the same beat based on the rules of the game [Figure 2.2].

By the same logic attributes in Beat Saber are x coordinate (four options), y coordinate (three options), rotation (nine options), and

hand (two options). Combining x and y coordinate gives 12 positions, each with 18 combinations of rotation and hand or unused. This results in $(18+1)^{12} \geq 10^{15}$ PDA per beat.

All of the possible combinations cannot be simply classified as was done for Dance Dance Revolution in previous research [3], because the one-hot encoding vector for such enumeration would need over 125 terabytes of RAM per beat.

Therefore we need to find a new approach of representing the actions. Such new data representations, their restrictions, and most fitting training types are discussed in the next subsections. Concrete examples of the different representations are shown in the Table 2.2. Influence of chosen data representation to model performance is evaluated in the Chapter 6.

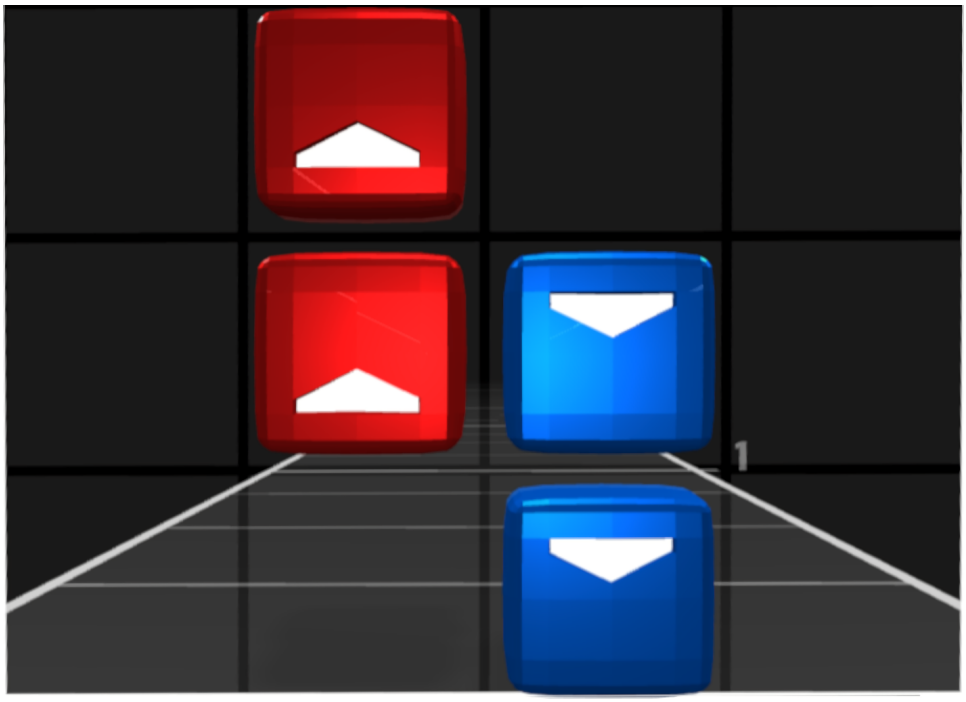

Figure 2.2: Example of action consisting of 4 beat elements from the view of the player. Bottom left corner has coordinates `x: 0, y: 0|`. `Top right corner has coordinates x: 3, y: 2|`.

2.2.1 Most used actions enumeration

The first approach to reduce the size of enumeration is to enumerate only actions used in the train set. We call such representation most used actions enumeration.

The first problem of this solution is the disability of the model to invent new actions. The second is the need for unknown tokens in the validation and test sets.

2.2.2 Action vector space

The shortcomings of most used actions enumeration can be omitted by embedding the actions into a vector space with semantically independent base vectors, if possible. We call such representation action vector space (AVS). Every action is represented as a point (k-dimensional vector) in the AVS. The ways to create AVS and its other usages are discussed in the Chapter 3.

2.2.3 Word bijection

To make most used actions enumeration and AVS easier to create, we define bijection between all possible actions and words. We call such representation word bijection. It is well defined by the functions `action2word` and `word2action` [Listing 2.1]: the beat elements are lexicographically sorted for word creation and action is a set of beat elements.

Listing 2.1: Word bijection definition

```
def _beat_element2word(be: BeatElement) -> str:
    return be.hand.name + be.x + be.y + be.rotation

def action2word(action: Action) -> str:
    words = [_beat_element2word(be) for be in action]
    sorted_words = sorted(words)
    return '_'.join(sorted_words)

def _word2beat_element(word: str) -> BeatElement:
    return BeatElement(hand=word[0], x=word[1],
                       y=word[2], rotation=word[3])
```

```
def word2action(word: str) -> Action:
    words = word.split('_')
    beat_elements = [_word2beat_element(w)
                     for w in words]
    return Action(beat_elements=beat_elements)
```

2.2.4 Other representations

Other tested representations are multi-label classification of each possible beat element placement separately and separate classification of each hand. Both achieve poor results in early development and are not further discussed.

Table 2.2: Action from Figure 2.2 in different representations

Representation name	Data example
Beat Saber JSON representation	`[ {x: 1, y: 1, rotation: up, hand: left}, {x: 1, y: 2, rotation: up, hand: left}, {x: 2, y: 1, rotation: down, hand: right}, {x: 2, y: 0, rotation: down, hand: right} ]`
Most used actions enumeration	`{ action_id: 793 }`
Action vector space	`[1.24, -2.47, 0.14, -2.12, ... ..., 0.16, 0.04, 0.19, 0.97, -1.42]`
Word bijection	`{ word: L110_L120_R201_R211 }`

2.3 Audio representation

Songs are stored in the Ogg Vorbis format with 44.1 kHz sample rate with different bit rates. The stereo audio is loaded into a NumPy array of shape `(frames, channels)` and the two channels are averaged

to generate a mono audio. We generate the Mel-frequency cepstral coe cients (MFCC) features as described by Suwajanakorn et al. [10] without the PCA. Using SpeechPy by Torfi [11]:

1. Apply pre-emphasis.

2. Take the Discrete Fourier Transform with a sliding window of `config.audio_processing.frame_length` (default 25) ms and `config.audio_processing.frame_skip` (default 10) ms sampling interval.

3. Apply 40 triangular Mel-filterbanks.

4. Apply log.

5. Compute Discrete Cosine Transform and take the first `config.audio_processing.num_cepstral` (default 13) cepstral coefficients.

6. Replace the first cepstral coefficient by a log of frame energy for to account for volume.

7. If `config.audio_processing.use_temp_derrivatives` equals `True`, add first temporal derivatives.

8. Shift the timestamps by `config.audio_processing.time_shift`.

The time shift is a technique proposed by Suwajanakorn et al. [10]. By shifting the audio, the model can condition based on the future without the use of bidirectional information.

To enable conditioning on rhythmic features, we add Δ-*time* as proposed by Donahue et al. [3]. Δ-*time* adds two features: the time to the previous and the next action.

For additional music context we propose adding *song part*. *Song part* is a single feature defined for each action as $\frac{\text{action.time}}{|song|}$. The model is trained from snippets, therefore this feature informs the model which part of the song it is generating.

2.4 Metadata representation

Metadata is stored in a info.dat file within each song folder. Metadata contains information about the song author, beat map author, Beat Saber environment for each difficulty, etc. The information needed for beat map creation is the initial beats per minute (BPM).

BPM can change during the song. The changes are stored within the JSON file for each difficulty as events of type 14. All changes under 30 BPM are disregarded as there are many low BPM mistakes in the datasets.

The original BPM is not stored, but used to recompute the JSON time in beats since beginning into seconds. Seconds are the only used time unit in the DeepSaber. As is pointed out by Donahue et al. [3], BPM changes make this non-trivial task. Performing the computation became the main bottleneck[5] of the preprocessing pipeline. The need to experiment with multiple hyper parameters for the dataset creation made us implement the time recomputation as a just in time compiled Numba function [12].

2.5 Beat map data structure and representation

To enable fast development, experimentation and avoid frequent recomputation, beat maps need to be stored in a different format than original JSON. Therefore processed beat maps are stored in a Pandas DataFrame [13] object, which is a common format for data science in Python. Dask[6] and Vaex[7] were considered, but dismissed for the reduced functionality. The datasets can fit into memory and parallelism is handled by processing each song folder independently, making most of the computation embarrassingly parallelizable.

Each beat map is split into snippets with a sliding window of length `config.beat_preprocessing.snippet_window_length` (default 50) beats and skip `config.beat_preprocessing.snippet_window_skip` (default 25) beats. The last window has smaller skip to exactly cover

5. In fact 97 % of the whole computation, resulting in ∼20 minutes to recompute one dataset on 12 thread machine. After optimizations this takes ∼30 seconds.
6. Dask: Scalable analytics in Python: dask.org
7. Vaex: Out-of-Core DataFrames: vaex.readthedocs.io

the last `config.beat_preprocessing.snippet_window_length` beats of the song.

Every row in the DataFrame represents one action of a beat map and it is uniquely indexed by a tuple of name of the song, difficulty, snippet number, and timestamp. Columns correspond to the action representations [Section 2.2] and audio information [Section 2.3]. For example DataFrame see Table 2.3.

Table 2.3: Preprocessed dataset as a DataFrame

				left_x	left_y	···	word	prev	next	···	part
name	**difficulty**	**snippet**	**time**								
Thriller	**Hard**	**0**	**6.2**	2	2	···	L228	0.0	0.7	···	0.01
			6.9	2	0	···	L208	0.7	0.4	···	0.02
⋮	⋮	⋮	⋮	⋮	⋮	⋱	⋮	⋮	⋮	⋱	⋮
		1	**27.2**	0	2	···	L026	0.7	0.6	···	0.21
⋮	⋮	⋮	⋮	⋮	⋮	⋱	⋮	⋮	⋮	⋱	⋮
	Hard	**0**	**6.2**	2	2	···	R228	0.0	0.3	···	0.01
⋮	⋮	⋮	⋮	⋮	⋮	⋱	⋮	⋮	⋮	⋱	⋮
Beat it	**Normal**	**0**	**7.1**	0	1	···	L010	0.0	1.1	···	0.01
⋮	⋮	⋮	⋮	⋮	⋮	⋱	⋮	⋮	⋮	⋱	⋮

2.6 Omitted data

The beat map JSON files include other information used for the Beat Saber gameplay. The most visually noticeable are lighting effects, bombs, and walls [Figure 2.3], all of which are omitted from DeepSaber model inputs, and outputs.

Once action selection is done, lighting effects can be added procedurally with LightMap[8]. Similar process could be used to generate bombs and walls.

8. LightMap: Automatic Lighting for BeatSaber custom maps: github.com/recrudesce/lightmap

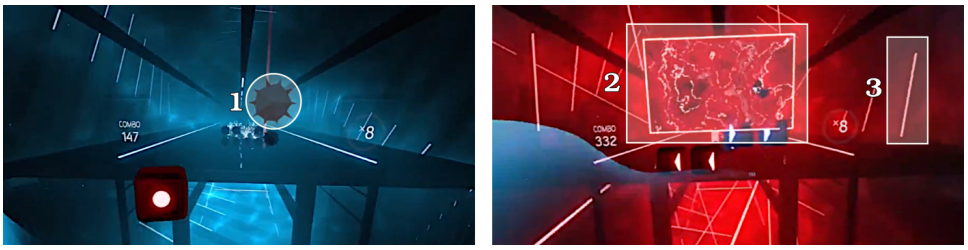

Figure 2.3: Other Beat Saber gameplay elements: bombs (**1**), walls (**2**), and lighting effects (**3**).

3 Preprocessing and data augmentation

To prevent overfitting and introduce more variance to the model, we consider multiple data augmentation methods. To prevent changing the original training data and keep the memory footprint low, the Keras Sequence[1] object is used. Sequence offers functionality to change the dataset at the end of each training epoch and to preprocess every batch given to the model. Compared to the Python generator, Sequence is safer for multiprocessing. Compared to more efficient `tf.data`[2], Sequence is more pythonic.

3.1 Action vector space creation

Important property of the Beat Saber gameplay is the flow [Figure 3.1]. The flow can be described as the natural swinging of the hands to the beat. The flow is rarely intentionally broken with the irregularity accompanied by increased Δ-*time*. The flow of the action at time t is mostly influenced by the actions at times $t \pm 1$, less by $t \pm 2$, etc. Therefore the flow is a local feature. The chosen action also depends on the general style of the song and other music features such as rhythm and part of the song. The current 10 seconds of audio need to be compared to the other parts of the song. Audio is a global feature.

The flow can be interpreted as the syntax of a sentence. Each action should move one or both hands into the roughly opposite position. If

1. Keras Sequence class: keras.io/api/utils/python_utils/#sequence-class
2. tf.data: Build TensorFlow input pipelines: tensorflow.org/guide/data

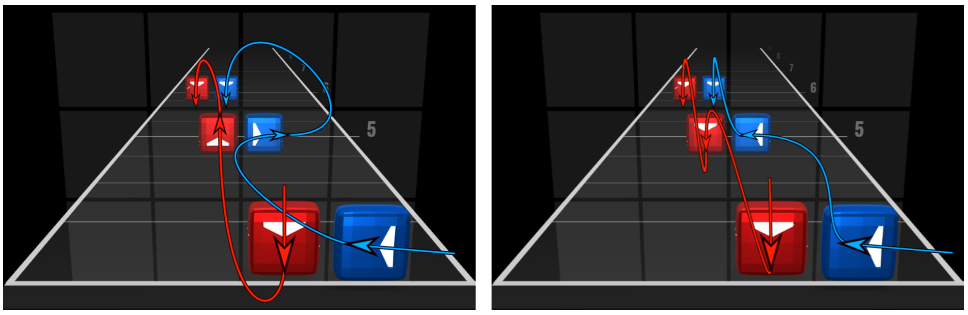

Figure 3.1: Comparison of good (**left**) and poor (**right**) flow. Added arrow lines show hand movement.

this rule is broken without enough Δ-*time*, the syntax is broken, and the beat map is hard to play.

3.1.1 Zipf's law for actions

Zipf's law was used to show the connection between linguistics and music (Zanette [14], Perotti et al. [15]) as a result of music generation with regard to the context. Beat Saber's actions show a similar fit to the Zipfian distribution as languages [Figure 3.2]. The Beat Saber dataset has fewer unique actions than Wikipedia's unique words. The exponent of action distribution is roughly 1.5 compared to 1.0 for Wikipedias.

3.1.2 Beat map as a sentence

Since actions follow Zipf's law, and the flow resembles syntax, we train continuous word representations that shown success in natural language processing tasks. Each beat map is turned into a short text of actions using word bijection [Subsection 2.2.3] and stored as a line in a corpus file.

3.1.3 Word vectors for actions

To use AVS representation, we need a function to assign each action in word bijection representation a vector. Formally, $vector : \text{string} \to \mathbb{R}^k$, where k is the dimensionality of the AVS. To obtain $vector$, we use unsupervised NLP techniques on the beat map corpus file.

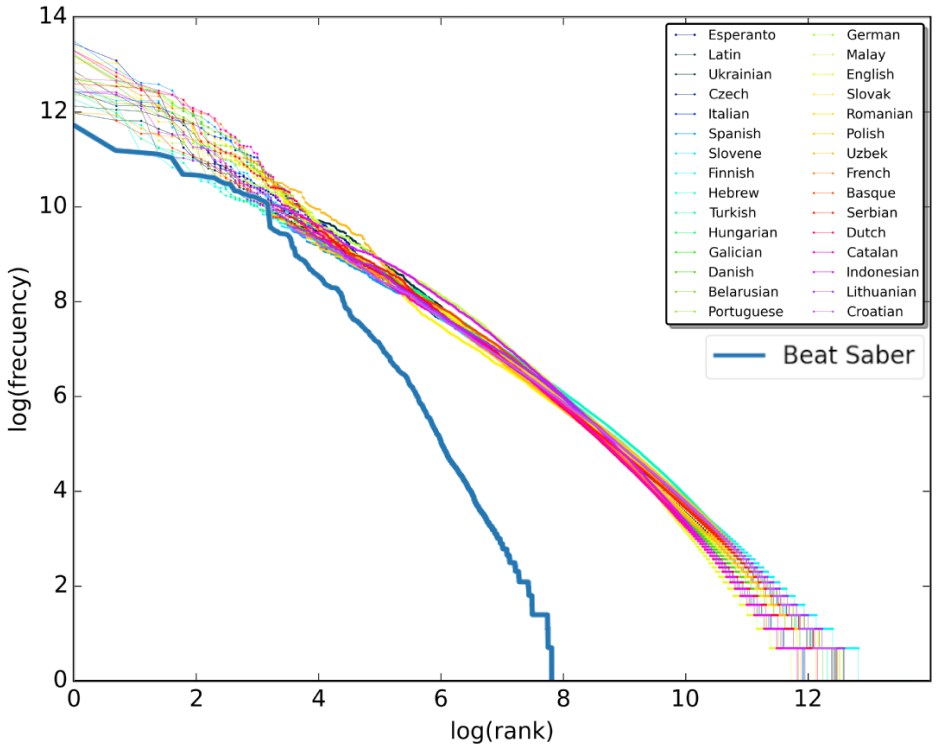

Figure 3.2: Composition of rank versus frequency in a log-log scale for the first 10 million words in 30 Wikipedias (dumps from October 2015) [16] with the 903,525 actions in OxAI train set.

There is not enough data to pre-train the state-of-the-art language models from the self-attention family (Polosukhin et al. [17]). We argue that because of the locality of the flow and a lack of polysemy, n-gram based models are sufficient to create high-quality AVS. This is later supported by the performance on our synthetic dataset and good correlation with accuracy when used as a metric for training.

We train the word embeddings with a continuous bag of words (CBOW) and skip-gram architectures [Figure 3.3] as proposed by Mikolov et al. [18]. The original hierarchical softmax objective benefits mainly when training on big corpora with a large number of classes (Joulin et al. [19]). The number of unique actions in Beat Saber datasets is a few orders of magnitude smaller than the number of unique words in the natural language dataset. Therefore we also evaluate the negative sampling objective as proposed by Mikolov et al. [20].

Morphology and semantics of actions in word bijection representation are highly connected. For example, the action `L000_R111` is literally the set of actions `L000` and `R111`. To use this sub-word information, we also train and evaluate the FastText model by Bojanowski el al. [21]. FastText is similar to Word2Vec with the distinction of representing words as a bag of character n-grams. For Word2Vec words `L000_R111`, `L000`, and `R111` are distinct vectors. For FastText with $n = 5$, they are represented as:

`{<L000, L000_, 000_R, 00_R1, 0_R11, _R111, R111>}`, `{<L000, L000>}`, and `{<R111, R111>}`. The words have overlap in the n-grams; therefore they have partially shared representation.

3.1.4 Synthetic action analogy dataset

To evaluate the quality of the action vectors, we create a synthetic action analogy dataset similar to the reasoning task defined by Mikolov et al. [18] [Table 3.1]. For example analogy pair (Athens, Greece), (Baghdad, Iraq) means Athens to Greece are like Baghdad to Iraq. Formally each analogy $(A_0, A_1), (B_0, B_1)$ pair is evaluated by computing $B_1^{pred} = vector^{-1}(vector(A_1) - vector(A_0) + vector(B_0))$[3] and compar-

3. $vector^{-1}(x)$ is word with the smallest cosine distance to vector x that was not used as the input.

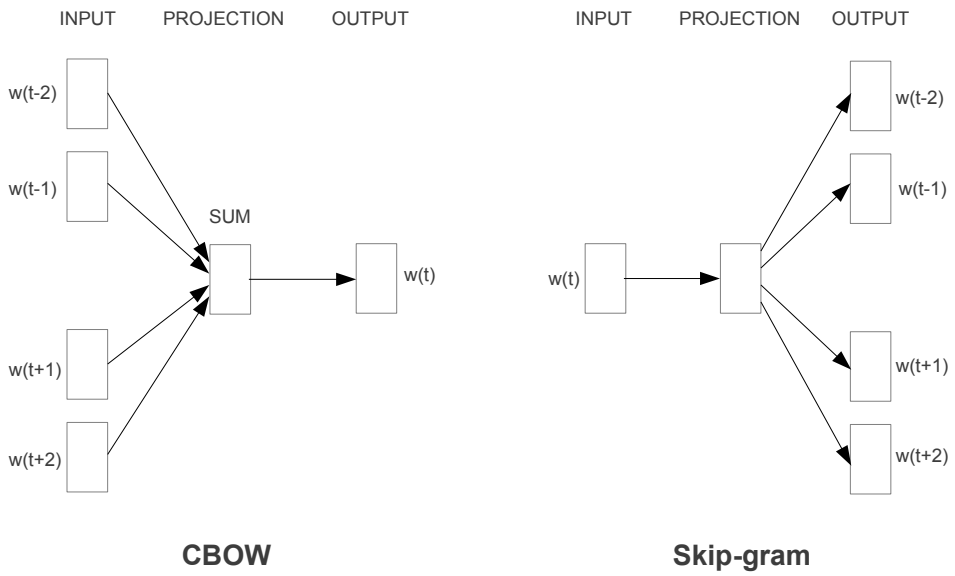

Figure 3.3: Word2Vec: The CBOW architecture predicts the current word based on the context, and the Skip-gram predicts surrounding words given the current word. [18]

ing B_1^{pred} with B_1. The top 1 accuracy is computed by comparing the prediction and ground truth for all of the analogies.

Table 3.1: Examples of three types of semantic and tree types of syntactic analogies in the Semantic Syntactic Word Relationship test set [18].

Type of relationship	**Word pair 1**		**Word pair2**	
Common capital city	Athens	Greece	Baghdad	Iraq
All capital cities	Abuja	Nigeria	Accra	Ghana
Family	boy	girl	brother	sister
⋮	⋮	⋮	⋮	⋮
Adjective to adverb	amazing	amazingly	apparent	apparently
Opposite	aware	unaware	informed	uninformed
Comparative	bad	worse	big	bigger

Our synthetic action analogy dataset consists of translation, rotation, hand swap, and double-beat in single hand and double hand versions [Table 3.2]. Where reasonable, the relationships between pair A and B are subset into *only x change*, *only y change*, and *both x and y change* groups for better performance understanding [Figure 3.4]. The reasoning[4] for introduced relationships is:

- Translation – the coordinate difference is independent of the absolute coordinate.
- Rotation – the rotation change is independent of position and the initial rotation.
- Hand swap – the left and right relations are independent of position and rotation.
- Double-beat – the coordinate difference between single and double beat element actions is independent of the absolute coordinate and rotation.

4. More information can be found in the create action embeddings notebook.

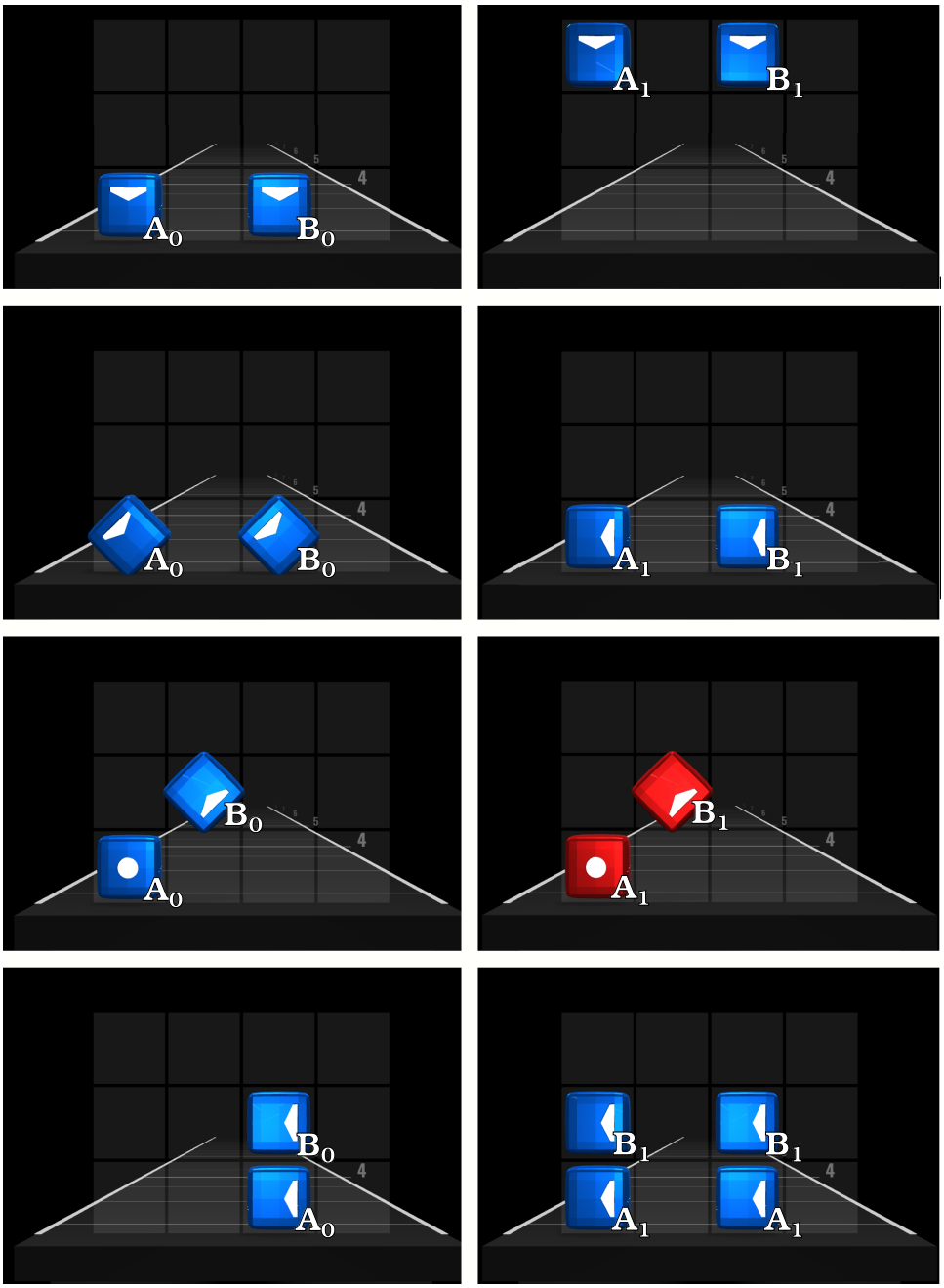

Figure 3.4: From top to bottom: single hand translation *only y change* (L000, L200), (L020, L220), single hand rotation *only x change* (L004, L003), (L024, L023), single hand hand swap *both x and y change* (L008, R008), (L117, R117), single hand double-beat *only y change* (L023, L003_L023), (L123, L103_L123).

Table 3.2: Three example pairs for each relationship type.

Type of relationship	Word pair 1		Word pair2	
Single hand translation	L030	L000	L130	L100
	L030	L000	L230	L200
	L130	L100	L030	L000
Single hand rotation	L008	L000	L018	L010
	L008	L000	L028	L020
	L008	L000	L038	L030
Single hand hand swap	L000	R000	L121	R121
	L000	R000	L122	R122
	L000	R000	L123	R123
Single hand double-beat	L032	L032_L002	L132	L132_L102
	L032	L032_L002	L232	L232_L202
	L132	L132_L102	L032	L032_L002
Double hand translation	L206_R101	L106_R001	L216_R111	L116_R011
	L206_R101	L106_R001	L226_R121	L126_R021
	L206_R101	L106_R001	L236_R131	L136_R031
Double hand rotation	L207_R008	L200_R001	L217_R018	L210_R011
	L207_R008	L200_R001	L227_R028	L220_R021
	L207_R008	L200_R001	L237_R038	L230_R031

3.1.5 Results

Because of the different nature of actions compared to natural language, we perform a hyperparameter search for both Word2Vec and FastText architectures. The models were trained on the OxAI train dataset, maximizing accuracy on the synthetic action analogy dataset. The search was done by bayesian global optimization with gaussian processes (Nogueira [22]). The best hyperparameters for each model type were evaluated by retraining on the OxAI validation dataset. Results shown in Table 3.3 omit double hand translation and rotation analogies for other than *x change* as there were zero, or near zero actions of that type[5] in the dataset.

The Word2Vec model has significantly better top 1 accuracy than random projection. The morphological information gained from *n*-

5. For example: L208_R100, L108_R000, L218_R110, L238_R038, or L230_R030.

Table 3.3: Synthetic action analogy dataset results when trained on OxAI validation dataset. Comparison of random projection (**Rand**), the best hyper parameters for Word2Vec (**WV** with CBOW), FastText with skip-gram (**FT SG**), and FastText with CBOW (**FT CBOW**). All the best models used negative sampling ($n = 5$), window size $= 1$, 3 iterations. Word2Vec used hidden layer of size $= 32$, FastText used hidden layer of size $= 256$ with added 2-grams and 3-grams.

	Rand	**WV CBOW**	**FT SG**	**FT CBOW**	**Size**
Type of realationship	**Accuracy [%]**				**[#]**
Single translation x	0.19	8.60	24.68	**36.91**	2592
Single translation y	0.24	7.29	32.76	**41.48**	1672
Single translation other	0.12	4.65	4.54	**7.61**	2468
Single rotation	0.10	5.23	65.51	**78.00**	11228
Single swap	0.15	1.52	**50.49**	43.33	13284
Single doublebeat y	0.00	3.70	56.79	**71.60**	80
Single doublebeat x	0.00	3.22	46.24	**61.29**	92
Single doublebeat other	0.00	4.64	53.16	**67.09**	236
Double translation x	0.00	15.52	24.84	**32.30**	160
Double rotation x	0.11	8.04	48.02	**52.66**	882
Total	0.13	4.16	49.03	**52.39**	32722
Average	0.09	6.24	40.70	**49.23**	3269

grams significantly increases the accuracy of the analogies. This can be explained by the way word bijection representation encodes the semantics into the syntax of a word.

3.2 Data augmentation

Basic approach to action augmentation would be to perform horizontal and vertical mirroring. This would not lead to good beat maps for humans, as the train dataset action distribution shows strong preference for the bottom line with action sequences starting from the right hand. The game offers an option to horizontally flip the maps for left handed people.

We use mixup data augmentation by Zhang et al. [23]. Mixup was shown to improve generalization on images, and sound, to stabilize the training of GANs, and to improve sentence classification (Guo et al. [24]). The idea is to generate new samples during training by linearly interpolating between random samples in each batch. Let (x_0, y_0) and (x_1, y_1) be samples from a batch. Let λ be a mixing ration drawn from a Beta(α, α) distribution, where α is a hyperparameter. The synthetic sample (x_g, y_g) is generated as

$$x_g = \lambda x_0 + (1 - \lambda) x_1,$$
$$y_g = \lambda y_0 + (1 - \lambda) y_1.$$

We also evaluate label smoothing to relax the confidence on labels as multiple actions are reasonable predictions. Let y be a one-hot encoded label vector and β be the smoothing parameter. The smoothed label y' is computed as

$$y' = (1 - \beta) y + \frac{\beta}{|y|}.$$

4 Deep learning architectures

"A machine learning algorithm is an algorithm that is able to learn from data. Deep learning is a specific kind of machine learning. Deep learning is a particular kind of machine learning that achieves great power and flexibility by learning to represent the world as a nested hierarchy of concepts, with each concept defined in relation to simpler concepts, and more abstract representations computed in terms of less abstract ones." [25]

Deep learning is an appropriate choice for high-dimensional, non-linear tasks, with plenty of annotated data; beat map generation is such a task. That is because the thousands of community created beat maps, the high-dimensional nature of sound, and the combinatorial explosion of possible actions.

4.1 Baseline

"Recurrent networks can learn from sequences and produce a sequence of states and outputs. One of the best-performing recurrent network architectures, the LSTM, propagates information through time via summation." [25]

As a baseline we use a 1-layer Long short-term memory (LSTM) [26] model. The model inputs are the current action in the most used dataset representation features, and the output is the next action in the most used dataset representation. Using a grid search, we choose 384 hidden units.

4.2 Dance Dance Convolution architecture

Donahue et al. [3] use a 2-layer LSTM model showed in Figure 4.1. The best accuracy and perplexity for Dance Dance Revolution are achieved with 64 hidden units and rhythmic features such as Δ-time. Beat Saber is a higher-dimensional problem; therefore, different number of hidden units is needed. We perform a grid search to find the optimum – 512 hidden units per layer.

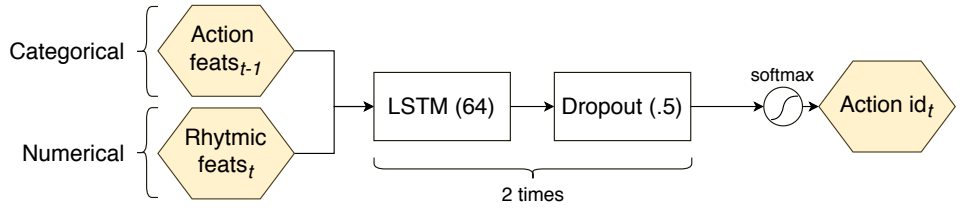

Figure 4.1: LSTM model with Dropout [27] used for action selection in Dance Dance Convolution [3]. Action feats is a bag-of-arrows representation containing 16 features (4 per arrow) to depict the previous step. Rhytmic features are Δ-time, or Δ-beat + beat phase.

4.3 Custom architecture

To see if there is an architecture similar to the DDC architecture, which works better with the extra information (AVS, audio features, difficulty), we perform a wider hyperparameter search. The hyperparameters for our custom architecture are shown in Figures 4.2, 4.3. The hyperparameter search is done using the Hyperband algorithm by Li et al. [28] implemented in Keras Tuner [29].

4.4 Multi LSTM architecture

The dimensionality of inputs varies widely. The optimal number of parameters required to express abstract representation from each input should be different. To find a good architecture addressing the difference in inputs dimensionality, we use a Multi LSTM (MLSTM) architecture, as shown in Figure 4.4. The connections between LSTM blocks are random; therefore, most of the architectures produce very poor results. We sidestep this issue by choosing a much larger than needed maximal number of epochs (110) in the Hyperband algorithm, resulting in many contestants in the first round.

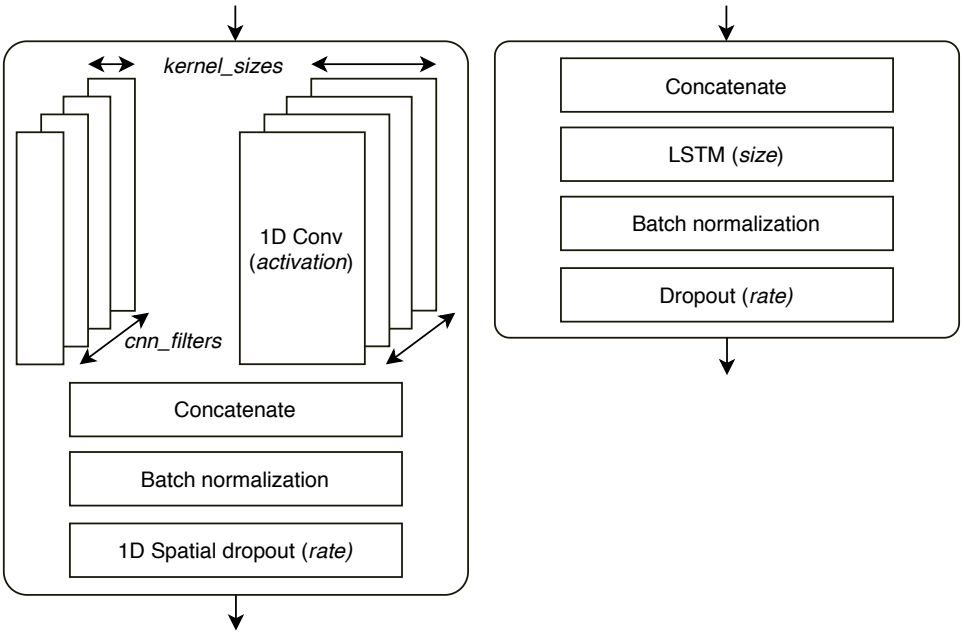

Figure 4.2: CNN block (**left**) and LSTM block (**right**). We use time-wise convolution (**1D Conv**) [25, Chapter 9], dropout [27], spatial dropout (feature-wise) [30], and batch normalization [31]. Hyperparameters in *italic*.

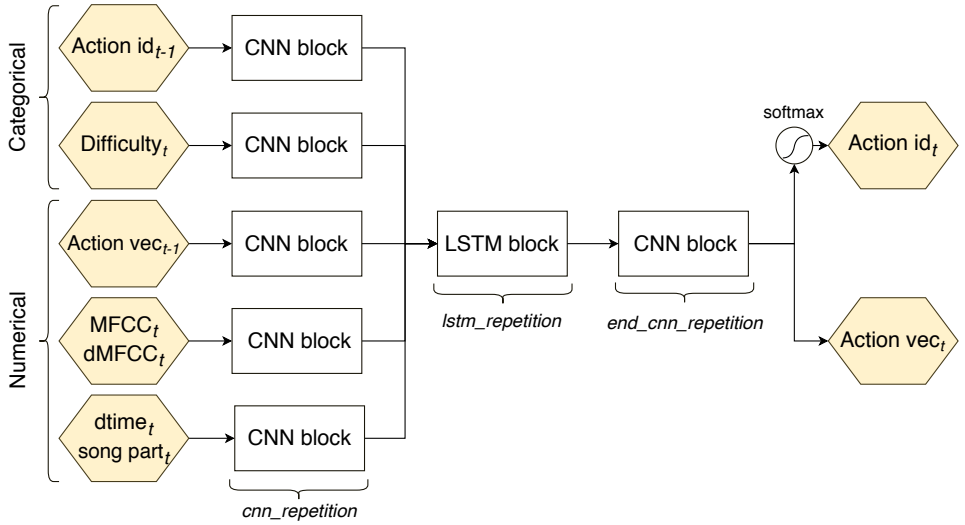

Figure 4.3: Custom architecture used for action selection. Hyperparameters in *italic*.

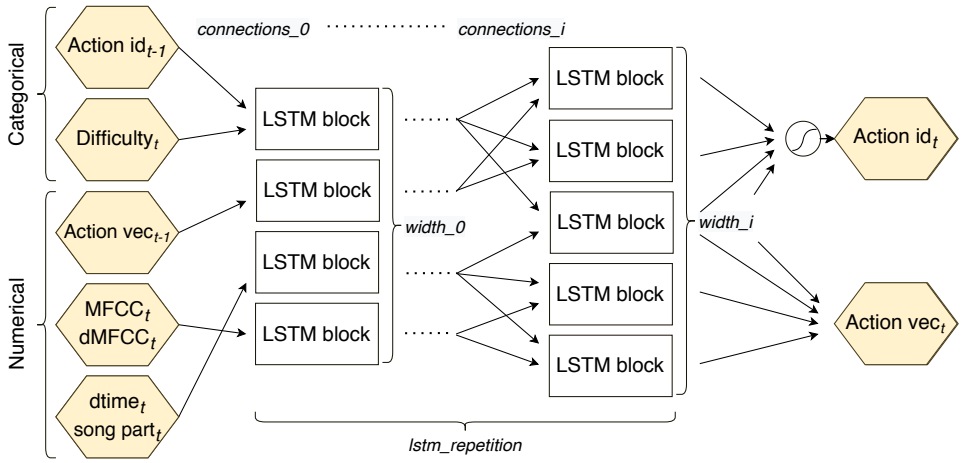

Figure 4.4: Multi LSTM architecture. Hyperparameters in *italic*. **lstm_repetition** is in range $\langle 2, 7 \rangle$, *connections* are in range $\langle 1, 3 \rangle$, and the *width* · *size* of any layer of LSTM blocks does not exceed 1024.

5 Metrics

"Determining your goals, in terms of which error metric to use, is a necessary first step because your error metric will guide all of your future actions." [25]

5.1 Dance Dance Convolution metrics

Metrics used by Donahue et al. [3] for action selection are average perplexity and average per-token accuracy with cross-entropy as a loss function. The same metrics and loss are used for language modeling, which is similar to beat map generation [Section 3.1]. Therefore, we use metrics perplexity (PPL), top 1 accuracy (Top 1 acc), and top 5 accuracy (Top 5 acc) calculated from the most used actions enumeration predictions (output action id) on per-token basis.

5.2 Action vector space distance

The problem with per-token accuracy for the generative model is the assumption of one correct prediction per beat. The quality of the gameplay would not change much if the action were moved by one block or rotated by 45°. We propose to use cosine distance in the previously created AVS as a metric for better interpretation of the model generative quality. Formally, given pair of strings y_{pred}, y_{truth} (actions in word bijection) we compute action vector space distance (AVS distance) as

$$\begin{aligned} \vec{y}_{pred} &= vector(y_{pred}), \\ \vec{y}_{truth} &= vector(y_{truth}), \\ \text{AVS distance} = \cos\text{distance}(\vec{y}_{pred}, \vec{y}_{truth}) &= 1 - \frac{\vec{y}_{pred} \cdot \vec{y}_{truth}}{\|\vec{y}_{pred}\| \|\vec{y}_{truth}\|}. \end{aligned}$$

5.3 Action velocity

All of the metrics so far measure only the quality of prediction for the next action given the sequence of previous actions used by humans.

Once we want to generate the beat maps for new songs, the model has to generate the next action from its own predictions. The model can learn to continue the current pattern and achieve high accuracy but never learns to start a new pattern. To measure the novelty of the generation, we propose action velocity. Given a beat map B, action velocity with window $= w$ at tth action is defined as

$$\text{action velocity}_{t,w} = \left\| \frac{B_{t-2w} + \ldots + B_{t-w-1} - B_{t-w} - \ldots - B_{t-1}}{w} \right\|.$$

Action velocity distribution (AVD) of a beat map B is defined as

$$B'_w = \{\text{action velocity}_{t,w} | t \in \mathbb{N}_0, t < |B|\}.$$

AVD of a set of beat maps S is defined as

$$S'_w = \bigcup_{B \in S} B'_w.$$

5.4 Action velocity distribution distance

We propose action velocity distribution distance (AVD distance) to measure the distance of two sets of beat maps. AVD distance is computed as the Kolmogorov-Smirnov statistic [32] between AVD of the beat map sets averaged over multiple window sizes. Given two sets of beat maps F, G, AVD distance of the first k windows is defined as

$$\text{AVD distance}_k = \frac{\text{KS statistic}(F'_1, G'_1) + \ldots + \text{KS statistic}(F'_k, G'_k)}{k}.$$

The AVD distance_{32} between the first 10^5 OxAI train actions and OxAI validation actions is 0.018. The AVD differs between difficulties (from $\text{AVD distance}_{32}(\text{Easy}, \text{Expert}) = 0.245$ to $\text{AVD distance}_{32}(\text{Expert}, \text{Expert+}) = 0.011$). Therefore, the generated and human beat maps are compared on on per-difficulty basis[1].

1. The generation of a beat map takes several minutes for both OxAI lab and Beat Sage; therefore, all comparison is done only on Expert difficulty.

6 Experiment evaluation

All experiments are done on the OxAI dataset for better comparison with OxAI beat map generation. Only normal, hard, and expert beat maps are used. Models with AVS as the output are minimizing mean squared error; all other models minimize cross-entropy. If not said otherwise, the data set was created using the default hyperparameters.

6.1 Per-token model performance

We train models as described in Chapter 4 with multiple hyperparameters and early stopping on AVS distance on validation data. For each model architecture, we select hyperparameters and the state which achieved the best AVS distance on the validation set and measure the performance on the test set with results in Table 6.1.

Table 6.1: Comparison of the best models for each architecture on perplexity (**PPL**), top 1 accuracy (**Top 1 acc** [%])], top 5 accuracy (**Top 5 acc** [%])], AVS distance (**AVS dist**), and time it took to train on the same machine (**Elapsed** [s]). Average of 7 complete train and evaluate cycles ± std. Models with action vector outputs have arbitrary (*abit*) perplexity based on the postprocessing.

	PPL	**Top 1 acc**	**Top 5 acc**	**AVS dist**
Baseline	19.3±0.16	33.1±0.04	63.8±0.13	0.556±0.000
DDC	16.3±0.35	35.0±0.14	66.8±0.18	0.541±0.001
Custom vec+id:id	24.4±0.53	37.0±0.15	68.6±0.14	0.524±0.001
Custom vec+id:vec	*arbit*	32.9±0.75	56.4±0.64	**0.308**±0.003
MLSTM vec+id:id	**14.8**±0.10	**37.3**±0.22	**69.3**±0.15	0.522±0.002

6.2 Influence of data sources

Except for the MLSTM architecture, the main difference between the models is the information in inputs and outputs. We evaluate the

influence of different inputs and outputs on a model with 2 LSTM layers of 512 hidden size. Results are shown in Table 6.2.

Table 6.2: Comparison of the influence of different input-output combinations. Average of 7 complete train and evaluate cycles ± std. **All** means all available input features were given to the model. Ordered by Top 5 acc.

Input		Output	PPL	Top 1 acc	Top 5 acc	AVS dist
Action	**Other**	**Action**				
	All	id	141.6±1.83	8.1±0.10	27.4±0.38	0.756±0.001
vec		vec	*arbit*	33.5±0.14	55.2±0.13	0.324±0.000
id		id	33.1±1.25	33.8±0.24	64.2±0.41	0.551±0.002
id	MFCC+dMFCC	id	32.0±0.69	33.5±0.29	64.2±0.25	0.554±0.003
id	MFCC	id	32.1±0.40	33.7±0.21	64.4±0.20	0.551±0.002
id	song part	id	32.1±0.64	34.0±0.11	64.6±0.10	0.549±0.001
id	difficulty	id	32.0±0.52	34.2±0.07	64.6±0.20	0.547±0.000
id	All	id	27.5±0.29	35.4±0.16	67.0±0.25	0.538±0.001
id	Δ-time	id	27.5±0.55	36.2±0.17	67.5±0.28	0.531±0.002
vec+id	All	id	27.1±0.71	36.4±0.17	68.0±0.41	0.530±0.002
vec+id	All	vec+id	**26.5**±0.28	**36.5**±0.17	**68.2**±0.36	**0.312**±0.001

6.3 Comparison of action selection

We compare the overall results of our approach and two similar projects using the AVD distance. All beat maps are generated on the same 10 songs on Expert difficulty[1]. For better comparison, our DeepSaber uses the Beat Sage action placement. For results see Table 6.3 and a commented video [Figure 6.1].

6.4 Reproducibility

The code to preprocess the beat maps, train AVS and run all of the experiments as well as the hyperparameters for the best performing models are available at github[2]. All training is done with RTX 2080ti,

1. Includes both beat-heavy songs, typical for Beat Saber, and beat-light songs like classical music. Concrete list on github.
2. DeepSaber repository: github.com/ronaldluc/DeepSaber

Table 6.3: Comparison of generation capabilities on the same 10 songs. **Human** is average of six sets of 10 Expert beat maps from human created test set ± std. Our models temperature was fine-tuned to minimize AVD $dist_{32}$ on the first 10 songs of train set.

	AVD $dist_5$	**AVD $dist_{12}$**	**AVD $dist_{32}$**
OxAI DeepSaber2	0.542	0.554	0.552
Beat Sage	0.154	0.150	0.133
Custom vec:vec	0.369	0.328	0.266
Custom vec:id	0.102	0.072	0.062
MLSTM vec+id:id	0.087	0.060	**0.042**
Human	**0.076**±0.019	**0.070**±0.020	0.077±0.016

and 32 GB of system memory using Python 3.8 Tensorflow 2.3 Docker image with CUDA 10.1. The default batch size is 128. To train one model takes between 700 and 1700 seconds. To run the experiments with a smaller footprint, smaller batch size (64), and a smaller portion of the dataset (50 %) can be used to achieve similar results.

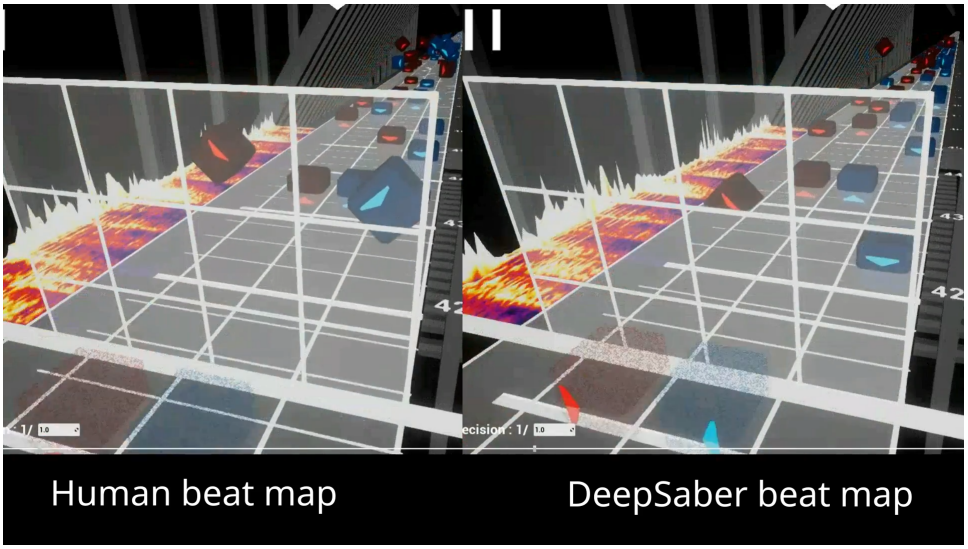

Figure 6.1: Side by side comparison of human created and DeepSaber generated beat map for the same song. DeepSaber uses the action placement used by human. The video uses a previous version of the model, without AVS. The currently best performing model (MLSTM vec+id:id) overcomes the shortcomings. YouTube: youtu.be/1rNNbtoovBM. FI MUNI: is.muni.cz/go/97hv0e

7 Conclusion and future work

In this work, we have described, implemented, and evaluated multiple approaches for Beat Saber map generation. We have shown the similarity between natural language and beat maps and used it to push the state-of-the-art in beat map generation. We have created an action analogy dataset and used it to demonstrate the high-quality of our action embeddings. We have used the action embeddings to create a new metric to be used during training and a new novelty based metric to compare fully generated beat maps with the maps generated by humans.

For the future plans, we will compute beat map embeddings using the unsupervised technique by Arora et al. [33]. We will assess the beat map quality by humans and measure the correlation between the action velocity distribution distance, beat map embeddings distance, and human judgment.

Another plan that is actively worked on is to wrap this thesis into an end to end application so that anyone can generate new beat maps at home.

Bibliography

1. MOORE, Ben. *The Best VR Games for 2019* [online] [visited on 2020-02-22]. Available from: https://www.pcmag.com/news/the-best-vr-games-for-2019.
2. MELCON, Andrew; PRIDAY, Richard. *The Best VR Games of 2020* [online] [visited on 2020-02-22]. Available from: https://www.tomsguide.com/best-picks/best-vr-games.
3. DONAHUE, Chris; LIPTON, Zachary C.; MCAULEY, Julian. Dance Dance Convolution [online]. 2017 [visited on 2020-02-22]. Available from arXiv: 1703.06891 [cs, stat].
4. O'KEEFFE, Karl. Dancing Monkeys (Automated Creation of Step Files for Dance Dance Revolution). 2003, pp. 66. Available also from: https://monket.net/dancing-monkeys/DancingMonkeys.pdf.
5. LIN, Zhiyu; XIAO, Kyle; RIEDL, Mark. GenerationMania: Learning to Semantically Choreograph [online]. 2019 [visited on 2020-02-22]. Available from arXiv: 1806.11170 [cs, eess].
6. LIANG, Yubin; LI, Wanxiang; IKEDA, Kokolo. Procedural Content Generation of Rhythm Games Using Deep Learning Methods. In: van der SPEK, Erik; GÖBEL, Stefan; DO, Ellen Yi-Luen; CLUA, Esteban; BAALSRUD HAUGE, Jannicke (eds.). *Entertainment Computing and Serious Games*. Cham: Springer International Publishing, 2019, pp. 134–145. Lecture Notes in Computer Science. ISBN 978-3-030-34644-7. Available from DOI: 10.1007/978-3-030-34644-7_11.
7. OXAI. *DeepSaber2 - BeatSaber Level Generator* [online] [visited on 2020-07-18]. Available from: http://oxai.org/2019/09/24/deepsaber2.html.
8. DONAHUE, Chris; AGARWAL, Abhay. *Beat Sage: Custom Beat Saber Level Generator* [online] [visited on 2020-07-18]. Available from: https://beatsage.com.

9. SMITH, Django. *Python Machine Learning: The Crash Course for Beginners to Programming and Deep Learning, Artificial Intelligence, Neural Networks and Data Science. Scikit Learn, Tensorflow, Pandas and Numpy.* S.l., 2019. ISBN 978-1-07-301933-5.

10. SUWAJANAKORN, Supasorn; SEITZ, Steven M.; KEMELMACHER-SHLIZERMAN, Ira. Synthesizing Obama: Learning Lip Sync from Audio. *ACM Transactions on Graphics* [online]. 2017, vol. 36, no. 4, pp. 1–13 [visited on 2020-06-27]. ISSN 0730-0301, 1557-7368. ISSN 0730-0301, 1557-7368. Available from DOI: 10.1145/3072959.3073640.

11. TORFI, Amirsina. SpeechPy - A Library for Speech Processing and Recognition. *Journal of Open Source Software* [online]. 2018, vol. 3, no. 27, pp. 749 [visited on 2020-06-27]. ISSN 2475-9066. Available from DOI: 10.21105/joss.00749.

12. LANARO, Gabriele. *Python High Performance - Second Edition.* [online]. Birmingham: Packt Publishing, 2017 [visited on 2020-06-27]. ISBN 978-1-78728-243-8 978-1-78728-289-6. Available from: http://www.myilibrary.com?id=1012761.

13. CHEN, Daniel Y. *Pandas for Everyone: Python Data Analysis* [online]. 2018 [visited on 2020-06-26]. ISBN 978-0-13-454704-6 978-0-13-454693-3. Available from: http://search.ebscohost.com/login.aspx?direct=true&scope=site&db=nlebk&db=nlabk&AN=1646871.

14. ZANETTE, Damian H. Zipf's Law and the Creation of Musical Context [online]. 2004 [visited on 2020-06-28]. Available from arXiv: cs/0406015.

15. PEROTTI, Juan Ignacio; BILLONI, Orlando Vito. On the Emergence of Zipf's Law in Music. *Physica A: Statistical Mechanics and its Applications* [online]. 2020, vol. 549, pp. 124309 [visited on 2020-06-28]. ISSN 03784371. Available from DOI: 10.1016/j.physa.2020.124309.

16. Zipf's Law. In: *Wikipedia* [online]. 2020 [visited on 2020-06-28]. Available from: https://en.wikipedia.org/w/index.php?title=Zipf%27s_law&oldid=964605848.

17. VASWANI, Ashish; SHAZEER, Noam; PARMAR, Niki; USZKOREIT, Jakob; JONES, Llion; GOMEZ, Aidan N.; KAISER, Lukasz; POLOSUKHIN, Illia. Attention Is All You Need [online]. 2017 [visited on 2020-06-29]. Available from arXiv: 1706.03762 [cs].

18. MIKOLOV, Tomas; CHEN, Kai; CORRADO, Greg; DEAN, Jeffrey. Efficient Estimation of Word Representations in Vector Space [online]. 2013 [visited on 2020-06-29]. Available from arXiv: 1301.3781 [cs].

19. JOULIN, Armand; GRAVE, Edouard; BOJANOWSKI, Piotr; MIKOLOV, Tomas. Bag of Tricks for Efficient Text Classification [online]. 2016 [visited on 2020-06-29]. Available from arXiv: 1607.01759 [cs].

20. MIKOLOV, Tomas; SUTSKEVER, Ilya; CHEN, Kai; CORRADO, Greg; DEAN, Jeffrey. Distributed Representations of Words and Phrases and Their Compositionality [online]. 2013 [visited on 2020-06-30]. Available from arXiv: 1310.4546 [cs, stat].

21. BOJANOWSKI, Piotr; GRAVE, Edouard; JOULIN, Armand; MIKOLOV, Tomas. Enriching Word Vectors with Subword Information [online]. 2017 [visited on 2020-06-29]. Available from arXiv: 1607.04606 [cs].

22. NOGUEIRA, Fernando. Bayesian Optimization: Open Source Constrained Global Optimization Tool for Python. 2014– . Available also from: github.com/fmfn/BayesianOptimization.

23. ZHANG, Hongyi; CISSE, Moustapha; DAUPHIN, Yann N.; LOPEZ-PAZ, David. Mixup: Beyond Empirical Risk Minimization [online]. 2018 [visited on 2020-07-03]. Available from arXiv: 1710.09412 [cs, stat].

24. GUO, Hongyu; MAO, Yongyi; ZHANG, Richong. Augmenting Data with Mixup for Sentence Classification: An Empirical Study [online]. 2019 [visited on 2020-07-03]. Available from arXiv: 1905.08941 [cs].

25. GOODFELLOW, Ian; BENGIO, Yoshua; COURVILLE, Aaron. *Deep Learning*. MIT Press, 2016. Available also from: http://www.deeplearningbook.org.

26. HOCHREITER, Sepp; SCHMIDHUBER, Jürgen. Long Short-Term Memory. *Neural Computation* [online]. 1997, vol. 9, no. 8, pp. 1735–1780 [visited on 2020-07-04]. ISSN 0899-7667. Available from DOI: 10.1162/neco.1997.9.8.1735.

27. SRIVASTAVA, Nitish; HINTON, Geoffrey; KRIZHEVSKY, Alex; SUTSKEVER, Ilya; SALAKHUTDINOV, Ruslan. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*. 2014, vol. 15, no. 56, pp. 1929–1958. Available also from: http://www.jmlr.org/papers/v15/srivastava14a.html.

28. LI, Lisha; JAMIESON, Kevin; DESALVO, Giulia; ROSTAMIZADEH, Afshin; TALWALKAR, Ameet. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization [online]. 2018 [visited on 2020-07-10]. Available from arXiv: 1603.06560 [cs, stat].

29. O'MALLEY, Tom; BURSZTEIN, Elie; LONG, James; CHOLLET, François; JIN, Haifeng; INVERNIZZI, Luca, et al. Keras Tuner. 2019. Available also from: github.com/keras-team/keras-tuner.

30. TOMPSON, Jonathan; GOROSHIN, Ross; JAIN, Arjun; LECUN, Yann; BREGLER, Christopher. Efficient Object Localization Using Convolutional Networks [online]. 2015 [visited on 2020-07-17]. Available from arXiv: 1411.4280 [cs].

31. IOFFE, Sergey; SZEGEDY, Christian. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift [online]. 2015 [visited on 2020-07-17]. Available from arXiv: 1502.03167 [cs].

32. Kolmogorov–Smirnov Test. In: *Wikipedia* [online]. 2020 [visited on 2020-07-17]. Available from: https://en.wikipedia.org/w/index.php?title=Kolmogorov%E2%80%93Smirnov_test&oldid=964433785.

33. ARORA, Sanjeev; LIANG, Yingyu; MA, Tengyu. A Simple but Tough-to-Beat Baseline for Sentence Embeddings [online]. 2016 [visited on 2020-07-11]. Available from: https://openreview.net/forum?id=SyK00v5xx.