Final Report

Project Background:

Rhythm games such as Dance Dance Revolution (DDR), Beat Saber, and Guitar Hero/Clone Hero only package a finite amount of levels with them. To play with songs not included in the original game, players would have to find and download community created levels, or create new ones themselves. Charting songs for rhythm games proves to take a decent amount of effort and technical know-how, making it inconvenient for non tech-savvy players to find or create levels for their favorite songs. Our project designs a system that generates charted songs for the three aforementioned rhythm games from raw audio inputs, so that players can automatically generate charted songs for any of their favorite songs more easily.

Project Design:

Our system consists of a distinct two step process, consisting of two machine learning algorithms. The first algorithm is intended to be shared across all games, and is meant to generate game agnostic files containing audio information from the input file. The second algorithm is game specific and it takes the generic files created by the first algorithm to produce game specific files containing the charted levels. This two step process allows us to combine the collective data from all three games for the first step, netting us better performance.

Audio Preprocessing:

Audio files may vary in ways that would throw off our neural networks to produce inaccurate models. For example, the audios may be recorded at different volumes, some may be noisy, the files could be in different formats, or different sample rates. Thus, it is important to preprocess our audio input files to normalize them before feeding them into the neural networks. Furthermore the representation of the songs within each game varied wildly, requiring preprocessing for each game to be done separately.

Our system handles DDR preprocessing using a Python library called Pydub, which uses ffmpeg to handle files of various formats. The files are loaded into our program as instances of AudioSegment, a class defined by Pydub. Then we called a Pydub function to reduce noises in these segments and normalized the volumes so they stay in a comfortable range. Finally, the program writes these processed audio data to a desired file format. In DDR's data set there is an unusual issue as well. Apparently some arcade cabinets have a custom song mode with a limitation that the song can't be more than 2 minutes long. However, the check is only for the header data of the file, not the actual file length, so people edited the headers to lie about the song's length. This caused significant problems with the audio handling libraries used before eventually being fixed by forcibly correcting all of the headers.

For Beat Saber's preprocessing, a *.egg* file was encoded with the Ogg-Vorbis codec, which required us to use the library PySoundFile to decode it. The note information was stored in a standard json format, but had to be processed specially as such. Audio data was normalized, as specified previously. The Beat Saber beat data was dependent on the sample rate of the song, and BPM. Due to difficulties trying to normalize different sample rate songs, only 44100hz songs were accepted for use in this project.

Clone hero preprocessing was also limited to 44100hz songs in order to maintain a consistent sample with the BeatSaber data. Because clone hero chart data contains information about BPM changes at specific time stamps, we were able to get very accurate scaling of the audio data.

BPM Calculation:

Accurate BPM information was needed in order for the networks to properly line up the beats. BPM was calculated with a layered sliding window technique, where the BPM of each moment is calculated with all the beats detected in the corresponding window. As the time interval progressed, each window was resized incrementally for the same timestamp in order to get as many possible samples and make the algorithm as efficient as possible. Our algorithm uses a variety of different statistical analysis over a subset of BPM values calculated using sliding windows of various lengths, to get the most accurate values.

First, each of the BPM values calculated at a time stamp where scaled relative to each other over several iterations. For each iteration, the scaling was applied to beats that were sufficiently below the threshold of the maximum BPM value at that timestamp. By scaling the BPM values in this way, we were able to account for BPM measurements that were factors of each other. For example, several sets of windows could caluclate a bpm of 45 for a timestamp, while another set of windows would calculate the BPM as 90 for the same timestamp. This could happen due to a numerous number of factors, the primary of which seemed to be irregular / changing bass lines and drum beats. In the given example, the 45 BPM values would be scaled to 90, and so the overall BPM of that timestamp would be assigned a value of 90 since the beats would no longer differ sufficiently from each other. In a more complicated example, adding a measured BPM value of 30 to that list would still give valid results because 30 would get scaled to 60, and 45 would be scaled to 90. In the next iteration, 60 would get scaled to 90 and the values would align. The end condition for the BPM scaling was a difference threshold at which a BPM was not considered a factor of a higher BPM value. In our tests, the largest value this could be while still providing accurate data but without massively increasing the size of the BPM values was 3 BPM. A fallback end condition for this scaling was when the minimum BPM value of the current iteration was greater than the largest BPM value of the original BPM measurements.

For each scaling iteration, we had to determine whether the current set of BPM values were similar to each other. The way we determined a sufficiently minimal spread of BPM for a timestamp was by calculating the standard deviation of the set of scaled BPM values at each iteration. This calculation was all done over several rounds of data modification. First, the scaled BPM values are sorted from min to max, and then in each iteration, the standard deviation is measured for the sorted

BPM values. If the result falls outside a specified threshold, The tail value of the sorted BPMs is strimmed, and then the standard deviation is calculated again. This process repeats until either there is only 1 BPM left, in which case there is no conclusive output for a timestamp, or until there are at least 3 BPM values that fall within the specified standard deviation. When the standard deviation was sufficiently low, the remaining BPM values of that timestamp are averaged and then output into a dictionary of measured BPMs per timestamp. In our trials, the standard deviation threshold that produced the most accurate BPM measurements was 5 standard deviations.

The methodology used produces satisfactory results on songs that have a consistent volume, and during sections with a lot of material, but is very lossy when calculating BPM for songs that contain long quiet segments. This loss is due to the fact that accurate beat detection relies on intermittent high volume beats. If the threshold for detecting loud beats is lowered too much, the calculated BPM for a window will be extremely high. This in turn massively increases the runtime of the algorithm. By tuning the volume threshold for beat detection this problem was somewhat alleviated, however the best approach was to use the previously calculated beat as the ongoing beat until a new measurement was reached.

BPM calculation was tested but not used for the Beat Saber implementation since the parameters were not needed for song playback, by normalizing all BPM to 60, and specifying note placement with respect to that normalization. Onset beat detection (discussed later) provided satisfying results with this normalized BPM, without loss of precision, due to how song playback in Beat Saber is handled.

Neural Network 1 - Audio to Generic / Beat Mapping:

This network had coordination issues as the various games each ended up needing separate outputs, as the generic data proved not to be as generic as we had hoped. While the file itself is generic, it encodes information about the difficulty of the chart and other game-specific information. These differences in each game made it clear that it was optimal to separate the networks to be game specific, though brief testing indicates that a unified network with modified inputs was plausible. However, due to the differences in speed of the games, an inherent property of how the player inputs notes, we found that there was no general best approach. We found that onset detection capabilities from existing audio processing packages generally did a much better job than attempting to process the audio data directly. A significant cause of this is the sheer amount of data present in audio. By reducing that data to manageable amounts, the networks are better able to find the information they need. The DDR network used two different onset strength functions (melspectrogram and constant-q spectrogram) of librosa as input as well as the zero crossing rate and rough volume (rms) instead of using the audio data directly. It also used the previously selected notes and current song information as input. The Beat Saber network used either the spectral flux onset function or the spectral difference onset function, which was very good at finding small sub-onsets, necessary for the fast two handed gameplay.

In testing, this network proved to be quite effective in finding onset beats, but in terms of gameplay, it was only slightly better than a network which would only select the final onsets determined by the aubio library. It did, however, prove surprisingly capable at determining when it should be placing jumps in DDR. Both this network and the simple placement strategy seem to have issues at the end of a song, being unable to determine that silence should mean no placements.

There were far too many songs to fit in memory all at once, so the data was split into groups and the model was trained on each group separately. Every epoch the groups would be shuffled together and reformed. While inside of a group the specific notes were in randomized order.

Since we had strong beat placement information at this stage, the network only evaluated at positions which could contain a placement. The generic data format we created has a maximum frequency of 1/192 of a measure (with 4 beats per measure). Many songs used lower frequency. If the song being trained on required such high frequencies, points of non-notes were randomly dropped to ensure reasonable data frequency between non-notes and notes.

The network has six numerical inputs along with the data derived from the audio. They are: time resolution (the frequency mentioned above), notes per second, jumps (2+ notes at the same time) per note, freeze arrows (notes hold down for long periods of time) per note, and a maximum simultaneous notes input which can range from 1 to 4. This last one was a major reason this network was not perfectly suited for the other games and was instead particular to DDR. In most DDR charts, there is a maximum of 2 arrows at once (a jump) but some charts go higher by requiring the use of hands or diagonal foot placement.

Neural Network 2 - Generic to Note Maps:

This network proved to be highly capable without even needing any audio information, just the previous steps and the time and distance between them. Initial attempts at audio information proved that the network would simply train to ignore the information entirely. However, the onset method used in the first network has promise. The network for that is currently still training and is not yet as good as the no-audio network, but it looks as though it is depending on the audio data for useful information and as such will likely achieve a better final score.

The network used the same 4-output each with 4 one-hot outputs that DDC used as inputs. While this output format loses some expressivity in that it cannot distinguish between different types of jumps as easily, it is significantly easier for the network to learn as there are no classes that are extremely rare. With the no audio network, using the sum of 4 categorical cross-entropy losses, I was able to achieve a total loss of 1.4. Interestingly up and down both had significantly higher losses than left and right. While this loss is quite high, manual testing indicated that the generated charts were mostly realistic with only a few outliers. There were networks that were able to get below that, but testing on validation data proved that those networks were overfitting, despite having over 6,000 songs as input data. This is massively more than DDC, which used less than 300 songs, but I was able to get even more. At the time I was only testing on songs which had a static BPM. After adjustments to allow for dynamic BPM, I had eleven thousand songs, which seemed to be enough to prevent overfitting. Each song has an average of 300 steps inside of it, though that number varies wildly depending on the song. Experiments with data augmentation suggested that simply having more songs was a more effective solution, though if the network started overfitting again I would likely use it again.

Notes in Beat Saber have 216 (technically 216^2, when considering two handed notes) possible orientations, in comparison to the 4 in DDR and 5 in Guitar Hero. Furthermore, there are significantly less custom maps (<1000) made for the game, since it was released very recently. As such, extensive prototyping was done to attempt to get the mappings to work in a satisfying manner. Note representation is made incredibly complex by the "flow" required in note setup. For instance, successive notes have to work in such a way that the player can physically slash the notes. An example would be that a block for the right hand that requires the player to slash down must be followed by a block in a similar location that is slicing upwards. Initial attempts resulted in mappings that were physically impossible to play, forcing crossovers and unnatural jerks.

For that reason, some sort of time sensitive algorithm had to be used in order to make sure that the successive notes were natural. For that reason, we attempted to use a LSTM (Long Short Term Memory) neural network to approximate the mappings. Unfortunately, even with extensive hyper parameter tunings, we were unable to get satisfactory results. This may be due to the high complexity of the note representations, or due to the small cohort of training data.

As a result, we resorted to a first order Markov model, and encoded the notes as 4 digit "words". This Markov model was time-agnostic but very effectively mapped the transition distributions from note to note. This allowed for very "flowy" mappings, that were occasionally a bit too fast to play comfortably. However, they never provided any impossible sequences of notes, and made some incredibly interesting patterns. This very simple method proved to provide the best gameplay experience.

Outcome:

Each game was playtested with a variety of songs from different genres. The performance of each was hard to gauge since there are no known metrics for our current problem. Some examples of the charts generated are shown below:

Beat Saber - ("Addiction" - Logic) - https://youtu.be/5Ks_qkYKADs

Implementation:

https://github.com/Crazychicken563/RhythmGameCharterAI

Related Work:

Simon Dixon, Onset Detection Revisited, in ``Proceedings of the 9th International Conference on Digital Audio Effects'' (DAFx-06), Montreal, Canada, 2006.

Jonhatan Foote and Shingo Uchihashi. The beat spectrum: a new approach to rhythm analysis. In IEEE International Conference on Multimedia and Expo (ICME 2001), pages 881884, Tokyo, Japan, August 2001.