Automatic Processing of Dance Dance Revolution

John Bauer

December 12, 2008

1 Introduction

The video game Dance Dance Revolution is a music-based game of timing. The game plays music and shows patterns of arrows synchronized with the music. The player stands on a platform marked with the same four arrows shown in the game and earns a score based on how closely his steps match the timing of the video game. A pattern of steps for a given song is called a step chart, and each step chart has a difficulty based on the speed of the music and the number of arrows.

Official versions and knockoffs of the game have several hundred songs in total. Furthermore, freeware versions exist that allow users to enter their own songs. To enter a new song in such a version, a user must find the exact tempo of the music and choose the timing of the arrows to associate with the music. In general, a good player will notice a discrepency of as little as 10 ms in the timing of a song, so the timing of the new song must be very precise.

This leads to three separate tasks that must all be completed to process a song for use with DDR. First, the correct tempo of the song must be found. Second, the music in each measure or beat must be analyzed to see what steps fit that section of the music. Finally, given this information, the program must assign steps in time to the music to produce a step chart of a given difficulty.

The goal of this project is to analyze a previously unknown song and produce a step chart compatible with one of the freeware versions of DDR.

2 Training Data

As previously stated, many official versions of DDR have been released, along with several knockoff games made by competitors. In addition, a large body of fan-written work exists for the freeware versions. Downloading this data from various websites gives an easy to find source of training data for this program.

However, there is a problem with the data available. Even the "official" songs are not officially released by Konami, but are ripped, timed and editted by fans. Worse, independently made fan work is often of questionable quality. Songs of both types are often completely unusable.

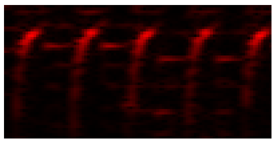
As it turns out, over the years I have collected some of this music for my own personal use. After eliminating some because they use features I am not working on here, I have over 350 songs timed to within the 10 ms accuracy discussed earlier.

3 Determining Tempo

The first thing to do is to find the tempo of a song. We can view the timing of the song as a function from the beat number to time elapsed from the start of the song. The tempo is then the derivative of this function. As the tempo may change over the course of a song, the derivative may not be constant or even continuous. What we do here is try to find a piecewise linear approximation to the timing function.

The first and most important aspect of finding the timing of a song is to find when a strong downbeat is. In fact, one can use this ability alone to calculate the tempo of a song. Once two strong beats have been identified, one can estimate the tempo in between the two by measuring the possible beats in between and using the tempo that gives the highest overall score.

One way to find a strong downbeat is to make a classifier that can recognize such a downbeat by calculating the spectrum of a song. To make such a classifier, we first need a set of features. The FMod API library is useful for this, as it can find the audio spectrum of a song. Using multiple frequencies from a several slices of time lets one build up a block of frequency strengths.



Spectra for a time close to the known beat (such as the one shown) are given a positive training label, and those for times in between beats are given a negative training label.

In fact, a procedure that seems to work well is one that actually treats the problem as two separate pieces. First, we look for a region in which we claim the nearest beat is within 50ms. Next, we look within that region for a smaller region in which the nearest beat is within 5ms. In both cases, positive examples are produced by taking the spectrum of a small window near a known beat, and negative examples are produced by taking the spectrum of a window outside the desired range.

The first classifier tried was that of Naive Bayes. This worked poorly in terms of precision/recall or other numerical measures. For example, on a training set of 3000 audio samples and a test set of 750, the "wide" classifier (50ms tolerance) had a training rate of 78.4% and a test rate of 76.4%. The "narrow" classifier (5ms tolerance) had a training rate of 70.0% and a test rate of 71.4% (random fluctuations are to be expected because of the songs used).

However, when applied to an actual piece of music, even these inaccurate results can often give good answers. For example, on a techno remix of "Xingfu de Ditu", by Elva Hsiao, this method acheived the 10ms tolerance discussed above. By scanning over 2s intervals at a time, first looking for the strongest match to the wide classifier and then looking in that region for the strongest match to the narrow classifier, the classifier accurately found the beats to within 10ms of the known good values.

Applying the routine to the song "Don't Sleep in the Subway" by Petula Clark, a song that does not have a constant tempo, led to a less satisfactory result, with the beats found being off by 20ms on average, although one beat was off by 50ms. However, the ground truth values for this song are noisy. The tempo is not steady, I had to time it by hand, and the rhythm of the song is much fainter than that of a thumping techno piece. All things considered, this is not a great result, but would be acceptable for casual use or could be corrected by hand for more serious players. (Interesting bit of trivia: the beat that is off by 50ms is one I always get wrong when playing this song; I assumed it was player error, but perhaps it is an error in the ground truth timing.)

On a song in which the classifier successfully finds accurate downbeats, finding the tempo that matches the intervening beats is easy. The tempo should be one that gives an integer number of beats between the two found downbeats, and the way to find that tempo is to take the one which gives the highest score using the same classifier.

An off-the-shelf SVM library, libsvm, produced a higher test rate: 87% on the "wide" classifier and 80% on the "narrow" classifier. This was trained with a linear kernel. I then used the grid search cross validation script that came with libsvm to look for a better Gaussian kernel, but the best one found had a test rate of 84% on the "wide" classifier test set. As it was larger, slower, and less accurate, I did not continue searching for better parameters for the SVM.

Also, I did not search for a polynomial kernel. My intuition, though, is that a good polynomial kernel would work well, as it could look for correlations between multiple parts of the spectrum that indicates how far apart the beats are coming.

One drawback to using the SVM is that the feature sets are very high dimension. The result is that the SVM is both large (300MB) and slow (roughly 1/10th the speed of Naive Bayes). This suggests a PCA approach, but such an approach is not implemented.

One specific case that can be improved is when the tempo is expected to be a constant tempo. This is often true for the songs used in DDR, such as techno or a lot of pop music. In that case, we can use linear regression to smooth out the pieces. In practice, this can work very well when the piecewise approximations were reasonably accurate. For example, on some songs, it can return a tempo within 0.01 beats per minute of the correct value, which gives errors of less than 10ms for the beats throughout the whole song.

This can go awry when one or more of the piecewise approximations are incorrect, though. One way to correct for this is to use outlier detection on the approximate tempos. I assume the tempo detection will take the shape of a Gaussian, with a mean where the true tempo is and a small random noise for the deviation. It turns out that assuming the measured tempos come from one Gaussian gives a closed form expression for the maximum likelihood Gaussian (no EM needed). I calculate this for the measured tempos and then ignore any tempos outside some predetermined range (in this case, 2 standard deviations).

This greatly improves the output in some cases. For example, one song from the official data is “Young Forever”, with a correct tempo of 140 BPM. The Naive Bayes approach with no outlier detection finds a BPM of 139.76, a serious error that even a novice player would notice. With Naive Bayes and outlier detection, the calculated BPM is 139.993, a very accurate result. Running the Naive Bayes algorithm without the outlier detection on a subset of the data gives a 38/145 accuracy rate for matching the tempo over the entire song. Using the outlier detection, the accuracy rate rises to 50/145.

One improvement that can be made would be to apply the outlier detection to songs that do not have a constant tempo. Other future directions include improving the usability of the SVM by using PCA and by trying a polynomial kernel.

A completely different approach, which was not implemented, would be to train a classifier that only accepts spectra of a certain tempo. We then find a strong beat and then take the spectrum of a couple second interval of the song in the region of that beat. The spectrum would then be expanded or compressed until the new classifier accepted it; the dilation needed would indicate the actual tempo of the song.

4 Step Timing

Once the tempo is known, the next task is to figure out where in the song the steps should go. Traditionally, if you know where the start and the end of a given beat are, there are six places a Dance Dance Revolution song can have a step. These are the downbeat, an eighth note later, one or three sixteenth notes after the downbeat, and either triplet after the downbeat. More than one step per beat is possible, of course. (Recent versions introduced additional timings, such as 32nd notes, but I ignored these for this project.)

The goal of this section is to figure out which subset of those six timings are eligible for a given beat. To build a step chart for a particular song, we can then use a greedy approach to add more steps until no more step times have an acceptably strong signal or until the steps produced meet a given difficulty criteria.

I tried three different approaches to find scores for the six timings, but none of them gave very satisfactory results.

The first method used features very similar to that used in the previous task, “Determining Tempo”. In this case, though, I considered the time range from just before the start of a beat to just after the start of the next beat in the music. These beats are then fed to a battery of Naive Bayes classifiers, one for each possible type of beat.

For example, it is very common in DDR step charts to have one step per beat, on the downbeat. One classifier out of the battery of classifiers takes as positive examples beats in which the downbeat is eligible for a step. Negative examples are beats in which the downbeat is not eligible for a step. Other beat patterns include pairs of eighth notes, an eighth note rest and an eighth note, and every imaginable subset of triplets or sixteenth notes.

As ground truth for this approach, I used the official step charts for the known songs that I have. If any of the possible step charts had a particular pattern for a given beat, I treated this as a positive training example. I also treated it as a positive training example for simplifications that would make sense to a user. For example, “1, and a 2, and a 3” can be simplified to “1 and 2 and 3” or to “1 2 3”, but not to “1, a 2, a 3”, which would sound unusual to a player. Otherwise, the beat was a negative example.

There are a couple problems with this approach. First, just because no step chart “realizes” a particular step pattern doesn’t mean it wouldn’t be suitable for that beat. For example, there are many songs available in which a sequence of eighth notes might be a reasonable pattern, but the ground truth step charts do not have those eighth notes. Accordingly, the classifier for eighth note runs will have many false negatives in its training data. Very rarely will there be a false positive in the training data, though. The result is that the trained classifiers are very conservative. Unfortunately, increasing the recall is not a solution, as the test data with negative scores near the threshold will be a mix of false negatives which were learned because of the bad training data and correct negatives which we do not want to turn into false positives.

Another problem is that many beat patterns are very rare. For example, the triplet pattern corresponding to swung eighth notes is relatively common: O_oO_o. . . The reflection of that pattern, Oo_Oo_. . . , does not occur anywhere in the set of songs I have.

Considering these problems, it is not surprising that the classifiers that were successfully trained only give an accuracy of 60% . . . 70%, and some classifiers that might be desired were simply impossible to train given the lack of data.

Another method I tried was to train one classifier that would give a score to any individual part of the step. The same classifier could be used to see if a step was suitable on the downbeat, on the eighth note after the downbeat, or any of the other possible subdivisions. Unfortunately, this classifier wound up learning the very simple rule of only ever accepting downbeats as a way of maximizing the training score.

I plotted precision-recall curve to see how to make it more effective, but the curve made it clear the amount of information in the classifier was rather low. Increasing the recall meant that too many false positives were mixed in with the new correct positives. As steps that are placed where they don’t belong are almost unplayable, it is important to keep a high precision; however, with a high precision, it is very difficult to get enough signal to place an interesting number of steps.

The third method I attempted, which was the one I finally used, was to train six classifiers that each tested one part of the subbeat. This method may actually be the “right” method, as it allows the step placement routine to distinguish between how important each individual part of the beat is. Unfortunately, it too suffered from the problem of unclean training data. The overall accuracy of the various classifiers averages out to 66%, not a very good result.

Once again, the fundamental problem seems to be with the data set. One of the problems in our learning theory problem set covered the idea of training in a setting where the truth labels are flipped with a random probability. Here, however, the flipping only ever occurs in one direction. This seems to cause the classifier to evenly mix the true negatives and the false negatives in terms of score.

The right solution would be to have an experienced player hand check the data and give the beat subdivisions labels. I haven't done this yet, as it would take several days to do for the current data set. My hope is to find some kind of distributed solution involving getting fans of the game online to help solve the problem, although I haven't done that yet, either.

5 Step Placement / Difficulty Assessment

Despite these problems with the step timing, the algorithm still comes up with interesting step patterns some of the time. When that happens, placing the actual steps is easy, although it might still be interesting to say a few words about that.

The basic assumption is that when a player is playing the game, there are some transitions in foot position that are easy to make and some that are harder to make. However, it is rare for a song to increase in difficulty by introducing harder foot patterns, as those simply aren't "fun" to play. The normal way to increase difficulty is to throw more and more steps at the player, still involving mostly "easy" transitions. I built a Markov chain using my own knowledge of the game in which the transitions from one foot position to another were given different weights based on how easy it was to make that transition. An interesting extenstion might have been to learn a HMM from the given data, but I did not explore this idea.

In order to avoid adding too many steps during this process, I used a linear classifier (least squares regression) to assess the difficulty of the song. Then, I greedily added steps by adding the step of the next highest signal from the step timing classifier until the linear classifier said the song was hard enough (a user parameter to the program).

The linear classifier used as features the number of steps in the part of the step chart with the highest step density. By measuring this over a couple different window sizes, this gave a reasonable description of the difficulty of a song. Once again, the ground truth data was the official DDR data, this time using the song difficulty labels for each step chart. This actually led to a high, but acceptable error rate. The classifier would rarely get the step chart difficulty exactly correct, but it would almost always be within 1 of the correct value.

A better way to do this would have been to use something such as libsvm's regression model on the same feature space, but this hardly seemed worth it considering the problems were elsewhere in the project.

One thing I did do to improve the step selection process was to run the step scores found in the previous section through a K-means algorithm. Then, instead of incrementally adding one step at a time, I took all of the steps at the same signal strength. The idea was that this would causes beats that sound similar to have the same step pattern, which would improve the overall quality of the step charts. It was hard to judge how effect this was, though; the problems described in the previous task meant that there not many finished products to compare with or without K-means.

6 Conclusion

The three step process described here gives the basis for a good method for making a previously unknown song compatible with (freeware versions of) DDR. Unfortunately, problems with the data used in the second step prevented the algorithm from being very successful. The hope is that building a new data set with more human input will give a program that does a credible job of creating new DDR step charts.

7 Acknowledgements

BemaniStyle: http://www.bemanisyle.com/

Libsvm: http://www.csie.ntu.edu.tw/ cjlin/libsvm

Stepmania: http://www.stepmania.com/

SVM light: http://svmlight.joachims.org/

FMod API, OpenCV, GFlags

Python, Visual Studio, Emacs, etc.

Professor Ng and the CS 229 staff