Automated game of Dance Dance Revolution using a neural network

Bobbie Smulders*
Artificial Intelligence Research Seminar
Media Technology MSc Programme
Leiden University, The Netherlands
(Dated: May 8, 2013)

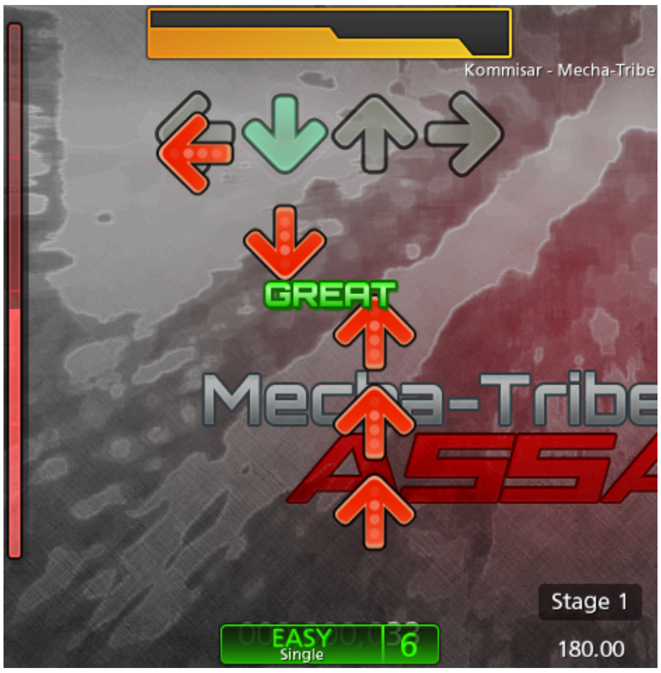
This paper describes the research on playing an automated game of Dance Dance Revolution using a neural network. It contains a breakdown of the game, a description of the experiments that were performed to get to the end result and an evaluation of the game played by a human, randomly, using a simple algorithm and using a neural network. Furthermore, we try too see if we need full knowledge of the game to be able to play it automatically. The conclusion is that it is possible to play the game using a neural network without having full knowledge of the game

Keywords: neural networks; feedforward; dance dance revolution; artificial intelligence; automated gaming

I. INTRODUCTION

The goal of the research is to answer two questions: Is it possible to use a neural network to play the game Dance Dance Revolution? And can we do it without having (full) knowledge of the game? A series of experiments will be performed to see if a neural network is up the task for this type of work, and if we indeed need full knowledge of the game.

Dissecting Dance Dance Revolution



To be able to automate a game of Dance Dance Revolution, it is important to know how it works. The game has a couple of important elements. The orange arrows on screen are notes. They are displayed by four arrows, left, right, up and down. These notes drift upwards at a rapid tempo. Whenever such a note is directly above one of the four fixed grey arrows, the player should press the corresponding button on his keyboard. This action is evaluated, and will result in one of the following texts being displayed on screen: “Flawless”, “Perfect”, “Great”, “Good”, “Bad” or “Miss”. Anything better than “Good” will also result in the note disappearing from the screen. In order to help the player, there is music playing in the background at the same rhythm as the key-presses.

The orange bar on top displays how good the player is. It will fill up slowly when a player continuously presses the right buttons, and will drain quickly when a player misses a note. The red bar on the left indicates the duration of the level and how far the player has progressed so far. The green block at the button states the difficulty of the game and the step timing.

Generalizing the game

The goal of this project is to play the game with minimal knowledge of the game. To do so, the game has to be viewed as general as possible. These are the rules that the solver application knows about:

1. The game has objects at certain two dimensional coordinates. These objects can and will move
2. It is possible to press four buttons. If an object disappears after pressing a button, it results in a higher final score
3. If pressing a button results in an object disappearing, it will cause the same effect if in the future another object is at the exact same coordinate.
4. An object can only disappear once during its life span
5. Pressing a button when it’s not necessary may result in lower final score
6. The goal is to make all objects disappear

* E-mail: b.smulders@umail.leidenuniv.nl

Previous research

There isn't any research to be found on this specific subject. However, there is a lot of research on using Dance Dance Revolution as exercise (especially on overweight children), but none of it tackled playing Dance Dance Revolution using a neural network.

II. APPLICATION SETUP

The first step was to create a solver application. This ended up as a Java application using a Swing GUI to inspect and manipulate the application. This application holds all the data for solving the game and can both "see" and "play" the game.

To interface with Dance Dance Revolution, it is possible to start the game and repeatedly take a screenshot to analyze the objects that appear on the screen. The problem with doing so is that image vision is quite tricky and that it takes up a lot of CPU (or GPU) time on high-resolution images. Because of the difficulty involved, and because it was out of the scope of the research, a different route was chosen.

Instead of using Dance Dance Revolution, the open source alternative StepMania was used. Being open source, it was rather easy to add a class to the codebase that sends the X and Y values of all the current onscreen objects (and the v-sync signal) over a network socket to the solver application. The same socket is also used to receive data from the solver application to simulate key-presses. This way, it was possible to create a very rapid screen-reading and a very rapid way of inserting key-presses into the game. Because of the rapid pace of the game, this is a crucial element.

III. DATA GATHERING

The first step in having the solver application automatically play the game is gathering data. This is done by playing a single level of the game and randomly pressing a button every 110 milliseconds. This way, we cover all buttons equally and can be sure that all possible reactions will be discovered. After a key-press is send to StepMania, every time a frame is drawn it is analyzed by a comparator algorithm. This algorithm checks the current frame and the last drawn frame. For every object on the last drawn frame, the algorithm checks for an object on the current frame that is roughly in the same position. It's fine if an object has slightly moved between the two frames, as the algorithm takes this into account. If an object on the last drawn frame doesn't have a corresponding object on the current frame, it means that it has disappeared.

As stated by the rules that the solver application know of (rule two and six), it is a good thing that an object disappears. So we label the object that disappeared with the button we just pressed ('1' through '4'). All the other object are labeled with '0'. After analyzing eacher object, we store the X position, Y position and the label in the data history array.

After playing one level (which takes about two minutes), we have enough data to work with.

An observant reader may have noticed that we do not account for objects that disappear because they leave the screen. First, the solver tries to figure out the border of the screen by analyzing the objects that appear on screen and constantly adjusting the border if an object appears outside the borders of the game. Then, the algorithm simple ignores objects that disappear near the border of the screen (using a five percent margin).

IV. USING THE SMARTH ALGORITHM

Before using the neural network to play the game, a very simple algorithm was written as a comparison to the neural network. This algorithm was given the name "smart", because it is an algorithm that is specifically tailored for this task. In hindsight, it might have been better to call this algorithm "dumb", as it has no intelligence and is rather simple.

The algorithm is called every time StepMania has drawn a frame. For every object on screen, it goes through the entire data history array (the one we created in the previous step). It looks for objects that have roughly the same coordinates (it can be 2 pixels off), and temporarily stores the labels belonging to those neighbors. After it is finished digging through the data history array, if one or more neighbors had a label with a value other than '0' attached to them, it looks for the most occurring label. After it has found this label, this label is stored. It then goes on to the next object on screen.

After going through all the objects in the current frame, we usually end up with one or two labels. The solver application sends the buttons linked to these labels to StepMania, and waits for 50 milliseconds before rerunning the algorithm.

This is of course no artificial intelligence. To use artificial intelligence, we introduce our application to a neural network.

V. USING THE NEURAL NETWORK

The solver application uses the Encog Machine Learning Framework. Using this framework, a simple feedforward neural network is created.

Neural network algorithm

The neural network algorithm is called every time StepMania has drawn a frame. For every object on screen, it inputs the X and Y coordinates into the neural network and it stores the label outputted by the neural network. If the resulting label is above '0', we store the label and go on to the next object. After going through all the objects in the current frame, we usually end up with one or two labels. The solver application sends the buttons linked to these labels to StepMania, and waits for 50 milliseconds before rerunning the algorithm.

Using a single output

With every neural network, getting the parameters right is crucial. At first, the most logical thing to do was to create a neural network with two inputs (the X and Y coordinates) and one output. The output would be between 0 and 1. By multiplying the output by 4, we recreate the original labeling system used by the data gathering step.

Fixing the lazy network

The problem that first popped up was that our neural network would get lazy. During the data gathering step, every time a frame is drawn (which happens 30 times per second), all the objects that did not disappear were labeled with '0'. Only when an object disappeared, it was labeled with the most recently pressed button. With the average level taking about 2 minutes, an average of 10 objects on screen, a frame rate of 30 frames per second and about 230 correct hits, this results in a dataset of about 36000 objects of which 230 are not labeled '0'. If the network always outputs '0', it is still 0.63 percent accurate. So the network would always output '0'. This quickly became a problem.

The solution for this was normalizing the data history. If we take the entire data history and categorize it by label, we for instance would get:

- Labeled '0': 17800 objects
- Labeled '1': 60 objects
- Labeled '2': 70 objects
- Labeled '3': 40 objects
- Labeled '4': 60 objects

In our final training set, we want a nicely balanced set of data points. So we first add every object labeled '0', then we add every object labeled '1' a total of 296 times (17800 divided by 60), every object labeled '2" a total of 254 times (17800 divided by 70) and so on. This gives us the following training set:

- Labeled '0': 17800 objects
- Labeled '1': $60\frac{17800}{60} = 17760$ objects
- Labeled '2': $70\frac{17800}{70} = 17780$ objects
- Labeled '3': $40\frac{17800}{40} = 17800$ objects
- Labeled '4': $60\frac{17800}{60} = 17760$ objects

Please note that we use integer division, as it is impossible to add an object 0.6 times to the training set.

Preliminary results

After finally getting data from the neural network, the data wasn't fully black and white. It was rather hard to make any sense of the data. A result of 0.88 could be rounded to 0.75 (meaning "press button 3") or could be rounded to 1 (meaning "press button 4"). Because of this, the neural network output was rather useless.

Using a seperate output for every possible button

At first, it seemed logical to only use a single output. Rule four clearly states that an object can only be "hit" by a single button. But because of the precision problem when using a single output, a quick experiment was set up using a neural network with four outputs. The inputs were still the same (the X and Y coordinates of each object in the data history array). The outputs mapped to the labels. If an object is labeled '0', the training set uses the outputs "0;0;0;0", if an object is labeled '2', the training set uses the outputs "0;1;0;0", if an object is labeled '4', the training set uses the outputs '0;0;0;1'.

The results of this experiment were very good. After inputting an X and Y value, the four outputs displayed a number between 0 and (roughly) 0.95. By using the output with the highest value and by using a threshold (0.86), it is possible to return a label corresponding to this location. If the third output gives the highest value, which (for example) is 0.91, it means that we should press button 3 right away. If none if the outputs are above 0.86, we shouldn't press any button. As stated before, we repeat this process for every object that is currently on screen so we know exactly which buttons to press.

Adding an extra output for doing nothing

The problem with having a threshold is that you either need to set this threshold to a specific value (which means that we tailor the neural network for this specific problem) or have a fancy method of finding the threshold value automatically. To solve this problem, an extra output was added to the network (with a slight bump in the amount of hidden layer neurons). The data input was slightly altered. If an object is labeled '0', the training set uses the outputs "1;0;0;0;0", if an object is labeled '2', the training set uses the outputs "0;1;0;0;0", if an object is labeled '4', the training set uses the outputs '0;0;0;0;1'.

Now it was just a matter of using the output with the highest value to know what to do: Either nothing, or pressing one of the four buttons. If all output values are zero, we translate this to the label '-1' (which also means: do nothing).

This didn't result in an increase in accuracy, but it did make the algorithm more generic, which is also an important part of the research.

Removing duplicate data

The data gathered during the data gathering step has a lot of duplicates. Not only that, it also has objects on roughly the same location with different labels. The reason this happens is easily explained: If button 2 is pressed and an object disappears, we label it '2' using the X and Y coordinates of the last known location. If, later on, we press button 1 when an object is in the exact same location, it will not disappear. Because of this, we end up with two objects with the same coordinates, one having the label '2', one having the label '0'.

To clean the data, each object is compared to every other object in the data history array. When two objects are close by, and have the same label, one of them will be removed from the training set. If two objects are close by and one of them has the label '0', the one with the label '0' will be removed from the training set. This action resulted in a slight accuracy bump and a much faster training session.

Final solution

Because of the high accuracy and speed of the experiments, the neural network with five outputs was chosen. There still are two inputs, namely the X and Y coordinates. There are two hidden layers in between, each having 25 nodes. Furthermore, we use the hyperbolic tangent activation function instead of the sigmoid activation function. Finally, the precision we want for our network is 0.5 percent with a training time limit of 10 seconds (although this constraint might need to be changed based on the available CPU time).

All data is normalized to prevent a lazy network. It is also stripped of duplicates to give a slight accuracy bump combined with a faster training session.

VI. EVALUATION

To answer the questions we asked ourself earlier ('Is it possible to use a neural network to play the game Dance Dance Revolution?' and 'Can we do it without having (full) knowledge of the game?'), we let the solver application play a game and compare it to a human player. Because we only want to know if it is possible to play a game and not how accurate it is, we only do one iteration per modus.

The game was set to normal mode, using the 'Mecha-Tribe Assault' song on easy mode (6 steps), using the following options: Little, No Jumps, No Hands, No Quads, No Stretch, No Rolls, No Lifts, No Fakes, No Holds, Mines off.

Results of a human player

The human player faired quite well:

Flawless	43 notes
Perfect	45 notes
Great	67 notes
Good	33 notes
Boo	13 notes
Miss	34 notes
Max combo	17 notes
Final score	675

Results of pressing random buttons

Pressing random buttons (something we do to gather the data needed for our neural network) isn't such a good strategy. This is clearly visible in the results:

Flawless	16 notes
Perfect	13 notes
Great	29 notes
Good	36 notes
Boo	40 notes
Miss	101 notes
Max combo	3 notes
Final score	331

Results of the smart algorithm

The smart algorithm played a superb game. It used the data history generated by the previous results (pressing random buttons) and therefore “learned” from its mistakes. The results:

Flawless	35 notes
Perfect	74 notes
Great	125 notes
Good	1 notes
Boo	0 notes
Miss	0 notes
Max combo	148 notes
Final score	848

It got a very impressive combo of 148 notes, meaning that it had a streak of 148 successfully hits without missing one note in between. It is also notable that there was only one “Good” note. This is because during the data gathering, we didn’t count “Good” as a successful hit (because getting a “Good” note doesn’t make it disappear). These results are a benchmark for the neural network algorithm. We are using the data history array in the most efficient way possible, so it is interesting to see how the neural network algorithm performs on the exact same data history.

Results of the neural network algorithm

After training the neural network and playing the game using the network, the results were very good:

Flawless	60 notes
Perfect	88 notes
Great	87 notes
Good	0 notes
Boo	0 notes
Miss	0 notes
Max combo	235 notes
Final score	913

The neural network played a “full combo”. This means that every note was hit, there were no “Good” notes, no “Boo” notes and no missed notes. It is very surprising that the neural network scored better than the smart algorithm, given the fact that the smart algorithm was tailored specifically for this task. One reason is that the smart algorithm scans for nearby neighbors in the data history array (which can be two pixels away). This results in the smart algorithm trying to hit a note very early on, as soon as a neighbor is in sight. That is why the smart algorithm has so many “Great” notes and one “Good” note, because the notes are basically hit too soon.

The neural network uses the same technique (roughly), but because it is not very precise (and even a bit reserved), it hits the notes a bit later on. This results in more “Perfect” and “Flawless” notes, resulting in a higher final score.

In all fairness, it should be noted that because the neural network is instantiated with random weights, the outcome will vary between sessions. About 10 percent of the time, the results are slightly hindered by a training session that has gone wrong. About 5 percent of the time the game is simply unplayable. For this session, the network was retrained until an acceptable outcome was reached.

VII. DISCUSSION

Is it possible to use a neural network to play the game Dance Dance Revolution? Yes, it most certainly is. The results of playing an actual level of the game using a neural network are a testament to this. The neural network performed better than an average human, better than pressing random buttons and better than an algorithm designed specifically for this task.

Can we do it without having (full) knowledge of the game? Yes, we most certainly can. During the development of the solver application, only the six rules stated in the introduction were taken into account. It was still possible to beat the game, even though we only worked with a very general description of the game.

From here one, there are a couple of possibilities to continue research on this project. It is possible to merge the data gathering step (pressing random buttons) with both training the neural network and playing the game using the neural network, by constantly training the network whenever an object disappears from the screen. There are two problems to this: What if (for instance) the network never tries to press button two? And what if we press button two and four simultaneously and one object disappears, which of the two button caused the object to disappear? The former can be solved by mixing both random keypresses and using the neural network. The latter might be a bit more complicated to solve. Further research might tackle these problems.

It is also possible to try to improve accuracy. Training the neural network currently goes wrong about 15 percent of the time. This is acceptable, because it can easily be retrained in a matter of seconds. But it does require a human to interpret the results, and it would be nice if we can remove the human intervention.

ACKNOWLEDGMENTS

I would like to thank Maarten Lamers for his solution on the problem of the lazy network and the suggestion on adding a fifth output to the neural network. Solving the problem of the lazy network was a very important step in the research, and the suggestion of adding a fifth output to the neural network (and thereby removing the need for a fixed threshold) made the solver more universal.