Universiteit Leiden
The Netherlands

Opleiding Informatica

Creating an AI for the Rhytm Game osu!

Jesper van Stee

Supervisors:
Dr. M. Preuss & Dr.ir. D.J. Broekens

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

19/08/2020

Abstract

An AI was created for `osu!` that performs two tasks, clicking and moving with some reductions in complexity. Overcoming some problems, the performance of the clicking task improved significantly over time in terms of accuracy. However, the functioning is still sub optimal after training. The performance of the moving tasks improved a lot in a rather short amount of time, with perfect accuracy achievable. However, there are still potential improvements to be made in both of the tasks.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Overview . . . 2

2 Related Work **3**

3 Background **4**
3.1 Gameplay . . . 4
3.1.1 Breakdown . . . 4
3.1.2 Objects . . . 5
3.1.3 Difficulty . . . 8
3.1.4 Scoring . . . 8
3.1.5 Modifiers . . . 9
3.2 Beatmaps . . . 10

4 Method **12**
4.1 Resources Used . . . 12
4.1.1 Shared Resources . . . 12
4.1.2 Additional Resources for Moving . . . 13
4.2 Data and Preprocessing . . . 13
4.3 Playing the Beatmaps . . . 15
4.4 Convolutional Neural Network . . . 15
4.5 Clicking Task . . . 16
4.6 Moving Task . . . 17

5 Experiments **18**
5.1 Clicking Task . . . 18
5.2 Moving Task . . . 21
5.3 Discussion . . . 24

6 Conclusions and Further Research **26**

References **27**

A Final Clicking Experiment Results Table **28**

B Final Moving Experiment Results Table **30**

1 Introduction

Over 17 million people play `osu!` worldwide. `osu!` is a fast-paced video game that falls into the category of rhytm games. The game is created by Dean Herbert and was released in 2007 [Her07]. The game receives regular updates and a new, open source, version called `osu!lazer` is currently being developed [Her16]. There are four different game modes: `osu!standard`, `osu!mania`, `osu!taiko`, and `osu!catch`. The most played game mode and the game mode that is relevant to this thesis is `osu!standard`, in general this is the game mode that people refer to when they are playing `osu!`, therefore in the rest of this thesis `osu!` will refer to `osu!standard`. Details of the gameplay will be discussed in a later chapter. The goal of this research is to create an AI for `osu!`, this takes us to the main research question of this thesis which is:

How hard is it to create an AI for "osu!"?

In the rest of this chapter the motivation of doing my thesis on this topic is given and there is an overview of the contents of this thesis.

1.1 Motivation

There are a few reasons that made me decide to work on this project. First of all, I enjoy playing video games, which made me want to do a project related to games. That in combination with my interest in machine learning got me looking into games I could make an AI for that would play the game. There are a few games I play regularly, but most of them did not seem feasible to create an AI for. However, I played `osu!` for a few years already, knowing well how to play the game and that the game is straightforward to play. Therefore, making an AI for `osu!` seemed to be within the realm of possibility.

1.2 Overview

In the first section after this some related works are discussed and in what way they relate to this thesis. In the section after that some background knowledge about the game is discussed. This is important to fully understand what is going on and why some decisions later are made. Then the method is explained and what resources are necessary for this approach. In the next section the results from the experiments are addressed. And finally the conclusions and possible further research.

2 Related Work

A lot of previous work has been done in the field of game AI. For `osu!` in particular, the most related work to this thesis was done by a user with the name *Vedal* on YouTube. He created an AI for the game that uses pixel data and a supervised learning method. The AI watches perfect playthroughs of songs in the game, then learn to play the game by mimicking the gameplay in the perfect playthrough[Ved14]. Unfortunately, no article has been written about this project for further details.

The AI for `osu!` will have two different tasks it needs to perform, two different methods will be used for this. Currently, a lot of research is being done in Deep Reinforcement Learning. One of the most significant works related to this thesis is the paper *Playing Atari with Deep Reinforcement Learning, 2013* by the DeepMind team[MKS+13]. They propose an algorithm called Deep Q-Learning, or Deep Q-Network (DQN), which is used in a part of creating an AI that will learn how to play `osu!` in this thesis. Their method uses an experience replay technique[Lin92] in combination with Q-Learning updates. The input they use for their neural networks is pixel data. The replay memory stores tuples of the state, which is the pixel data, the action taken, the reward earned, and the next state. Samples from the replay memory are randomly selected and then Q-learning updates are applied to the neural network. A random action is selected and executed by an ϵ-Greedy policy, if the value is smaller than ϵ perform a random action, otherwise perform the action from the neural network output. Directly related to this paper is a blog post by *Neven Pičuljan* using DeepMind's approach to create an AI that learns how to play Flappy Bird[Pi8] with very good results.

The related works that sparked most interest in this topic for me, are found on YouTube. For example *SethBling*'s channel[Set06]. He created different AIs himself, for a few different games, for some only pixel data is used. Notably he made an AI for Super Mario World and Super Mario Kart, the Mario Kart AI uses the same technique that DeepMind proposed.

3 Background

In order to create an AI for `osu!` it is essential to understand how the game works and what information is accessible. In this chapter that information will be discussed in a gameplay section and where it comes together in a beatmap file.

3.1 Gameplay

`osu!` is fairly straightforward in it's core, gameplay consists of performing moving and clicking actions on objects that appear in the gameplay window. The most common ways of players performing these actions is by using a mouse and keyboard, or a drawing tablet and keyboard. What objects and where they appear is defined in a beatmap that are created by the playerbase. The intensity of section within a song often decides the difficulty of gameplay in that section and the beatmap overall.

3.1.1 Breakdown

Figure 1 contains a breakdown of information displayed on screen during normal gameplay. See below the meaning of each marked component.

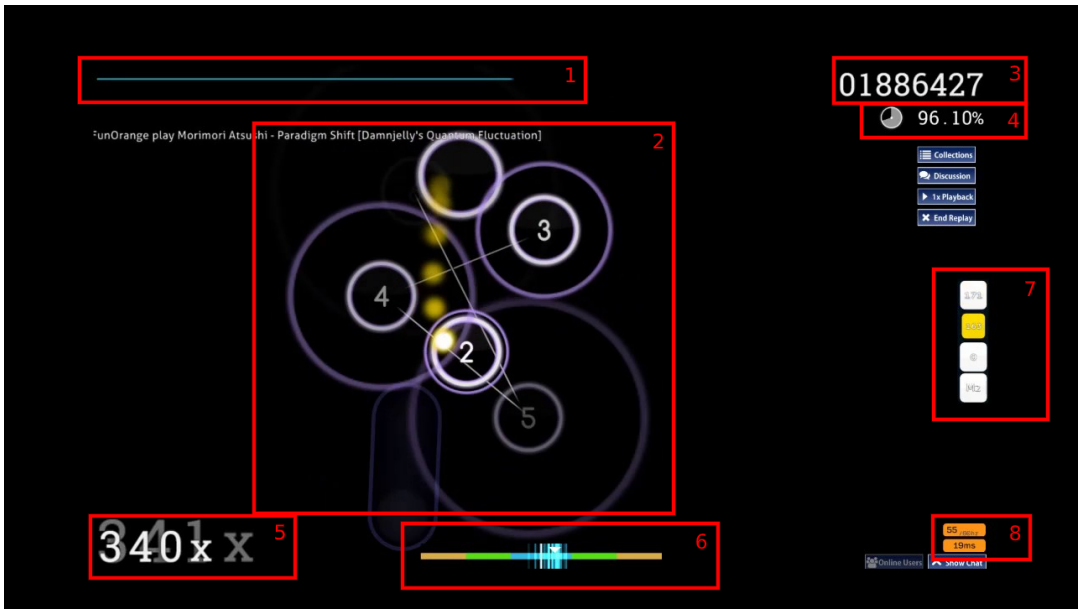

Figure 1: Breakdown of the gameplay window during a replay.

1. Health bar, affected by a passive drain rate and a penalty for miss timing an object. The player fails the beatmap if this reaches zero.

2. Hit objects and cursor, the objects a player is supposed to click at the right timing. The player just clicked the object that would have been number 1 and moves to the object with number 2. The ring around the object indicates the timing.

3. Score, a calculation of all objects the player has hit.

4. Accuracy, the percentage of objects that were hit with a perfect timing by the player.

5. Combo, the number of objects hit consecutively.

6. Hit error, a visualization of timing inaccuracies. The white bar indicates the perfect click timing. Hits within the blue section give a perfect score of 300 for the hit and 100% accuracy. Hits within the green section a score of 100 and 33% accuracy. And hits within the orange a score of 50 and 16.67% accuracy.

7. Keypresses, there are four possible keys a player can use to click the objects, from top to bottom: keyboard key 1, keyboard key 2, mouse button 1, mouse button 2. Most players use keyboard key 1 and keyboard key 2 as Z and X respectively. Mouse buttons are generally unused.

8. Framerate and frametime, due to the nature of the game and it's pace players like to display these values. A high framerate and a low frametime is important to be able to hit objects consistently within the perfect timeframe.

3.1.2 Objects

As mentioned before the objective of the game is to click objects that appear in the game window at the best possible timing. There are three types of objects in the game that are chained together to give a feel and shape to a beatmap.[faqnd]

- Hit circle, the simplest object in the game. The player taps and then releases one of the keys they use on top of the circle. The circle to hit is the one that has the smallest outer circle around the object. The best timing of the click is when the outer circle is as close as possible to the white border shown in Figure 2.

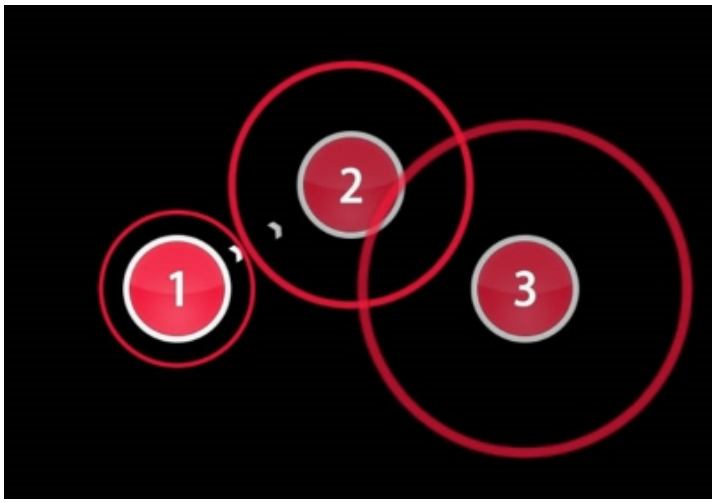

Figure 2: Three hit circles meant to be hit in order 1-2-3.

- Slider, this object requires the player to tap a key at the beginning circle of the slider at the correct timing, similar to a hit circle. Instead of releasing the key the player holds the key and follows the ball that appears until it reaches the end of the slider. Some sliders have an arrow indicating to follow the slider again but in reverse direction, this can occur multiple times with one slider. During the rolling the player receives a small score increase for every tick on the slider. Once the end of the slider is reached release the key and a score is awarded. An example slider with a return arrow is shown in Figure 3.

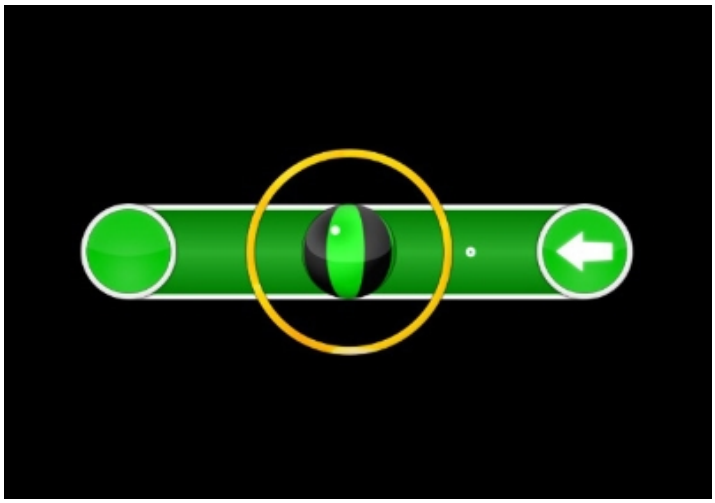

Figure 3: A slider with an arrow indicating to follow the slider in reverse direction back to the start.

- Spinner, requires the player to hold the keypress and move the cursor in either clockwise or counterclockwise direction to spin. Spin until at least the indication that the spinner is completed to not reset the player's combo. If the spinner is completed early the player can receive bonus points for additional spins. Figure 4 shows a cleared spinner with 40 000 bonus points achieved.

Figure 4: Cleared spinner with 40 000 bonus points for spinning after completion.

3.1.3 Difficulty

Every beatmap has a star rating, a higher star rating means a more difficult map. In `osu!` there are several factors that affect the difficulty of a beatmap. The biggest factor is placement and spacing of objects, generally higher spacing means higher difficulty. When creating a beatmap there are several variables that have an effect on the objects and affect the difficulty.

- Health Point drain rate (HP). This affects the rate at which health points drain throughout a song. Recovering health points from penalties becomes more difficult.
- Circle size (CS). Affects the size of hit circles and sliders. A higher circle size means smaller circles, therefore hitting the circles becomes harder.
- Approach rate (AR). The speed object appear at. A higher number means less reaction time to hit an object.
- Overall difficulty (OD). How difficult it is to hit objects in the perfect timing window. A higher value increases the difficulty.

3.1.4 Scoring

Every cleared object grants a score. The score of hit circles and the end of slider is calculated with the following equation:

$$Score\ Increase = H + (H \times \frac{M_{combo} \times M_{diff} \times M_{mod}}{25})$$

Where H stands for Hit Value, the score awarded from a hit (300, 100, or 50) and slider ticks and spinner bonuses.
M are multipliers from combo, difficulty and mods.
Combo multiplier is equal to $(Combo\ before\ hit - 1)$ or 0.
Difficulty multiplier is determined by the beatmap's variable settings. Add up all the values of each difficulty setting. The difficulty multiplier for each range is shown in Table 1.

Sum of points	Difficulty multiplier
$0-5$	$2\times$
$6-12$	$3\times$
$13-17$	$4\times$
$18-24$	$5\times$
$25-30$	$6\times$

Table 1: Difficulty modifier for different ranges of sums of difficulty values.

And finally mod multiplier is a multiplier based on selected mods. See the next section: section 3.1.5 Modifiers for the mods and multipliers.

3.1.5 Modifiers

Modifiers or in short mods are used to increase difficulty, decrease difficulty, or have a special function. Most modifiers affect the score players receive. Some of the modifiers can be useful for the development of an AI as will be described later.
The difficulty increasing mods are:

- Hard Rock (HR). Increase circle size value by 30%, so circles become smaller, and increase other difficulty settings by 40%. Score multiplier of $1.06\times$
- Sudden Death (SD)/Perfect (PF). Sudden death means that if the player misses a note they fail the beatmap immediately. Perfect is a step up from that, receiving a less than perfect score for an object will make the player fail. No score multiplier changes.
- Double Time (DT)/Nightcore (NC). Increases the speed of the beatmap's Beats Per Minute (BPM) to 150%. Song length becomes 33% shorter. Nightcore changes the pitch and adds a drum track to a song, other than that is the same as double time. AR, HP, and OD are slightly increased in value. Score multiplier of $1.12\times$.
- Hidden (HD). Removes approach circles and makes hit objects fade away after appearing on screen for a short period. Score multiplier of $1.06\times$.
- Flashlight (FL). Limits the visible area during gameplay to a small area around the cursor. Score multiplier of $1.12\times$.

Difficulty reducing modifiers:

- Easy (EZ). Decrease all difficulty values by 50% and the player gets 2 additional lives when the health bar reaches zero. Score multiplier of 0.50×
- No Fail (NF). The player cannot fail. If the health bar reaches zero the player can continue playing. Score multiplier of 0.50×.
- Half Time (HT). Decrease the speed of the beatmap's Beats Per Minute (BPM) to 75%. Song length becomes 33% longer. AR, HP, and OD are slightly decreased in value. Score multiplier of 0.30×.

Special modifiers:

- Relax (Rx). The player does not need to click, only moving is necessary. The player also cannot fail and spinning Rotations Per Minute (RPM) is doubled. Score multiplier of 0.00×.
- Auto Pilot (AP). The player does not need to move, only clicking is necessary. The player cannot fail with this mod either. Score multiplier of 0.00×.
- Spun Out (SO). Spinners will be automatically completed. Score multiplier of 0.90×.
- Auto. When turned on it shows an automated perfect playthrough of a beatmap.

Many combinations of mods are possible like Hard Rock, Double Time, Hidden and Flashlight combined will give a multiplier of 1.41×. However some combinations are not compatible like Hard Rock and Easy, or Double Time and Half Time.

3.2 Beatmaps

Something not directly useful to have knowledge of for regular players is to understand how beatmap files are saved. Beatmaps are stored in `.osu` file format, which is readable for humans. A beatmap file has it's content split up into different sections. Some general information like `.mp3` filename, a timestamp for preview etc. are stored under General. Editor information is relevant for editing the beatmap. Metadata contains the artist name, song title, and beatmap creator. The difficulty section has the values from

section 3.1.3 Difficulty stored. Then the Event and TimingPoints sections do not have much useful information for creating an AI. Neither is the Color section. Finally the most useful information stored in a beatmap file is under HitObjects.

HitObjects contains information about what object appear on screen at what time. Each line is comma seperated following this syntax:

```
x,y,time,type,hitSound,objectParams,hitSample
```

`x` and `y` are the coordinates within the game window. `Time` is the time an object is supposed to be hit in milliseconds from the start of the song. `hitSound` what sound the object makes, `hitSample` is related to the sound of an object as well. Sound information is not useful information to be used in this thesis so will not be elaborated on. `ObjectParams` contain extra parameters for some objects, for example sliders have the following syntax in place of `objectParams`:

```
...,curveType|curvePoints,slides,length,edgeSounds,edgeSets,...
```

Where `curveType` indicates the shape of the slider and `curvePoints` are pipe-seperated coordinates of the form `x:y` each coordinate stands for a point that gives shape to the slider. `Slides` means the number of repeats of the slider, `length` the length in pixels on screen. `edgeSounds` and `edgeSets` have to do with audio.

Spinners only have `endTime` as objectParam. Combined with `time` you can find out the length of a spinner.

4 Method

In the fourth chapter, the method of creating an AI for `osu!` is discussed. One of the first decisions made is to split up the AI in two main task. One task is learning to click and the other task is learning to move. Before continuing on the method for each of these tasks, the resources needed will be discussed.

4.1 Resources Used

There are many resources used for the different tasks, most are shared between the two and a few are specifically used for one task only. First the shared resources will be discussed. Later within this section the additional resources for the specific tasks are mentioned.

4.1.1 Shared Resources

One of the most important resources used is the Python programming language. Python offers many packages and is widely used for machine learning, therefore very suitable. Python version 3.6.6 is used. A variety of modules were used, below a list of the most important modules that both approaches use and what they are needed for.

- PyTorch version 1.4.0 is used for the neural networks.
- OpenCV version 4.2.0 is used to apply image transformations easily.
- mss is used for the screenshots of the game window.
- win32 is used for memory operations.
- ctypes provides C data types used for memory operations.

The system used to run the experiments is a laptop with an `Intel i7-9750H CPU` running at `2.60 GHz` and a `GTX 1660 Ti` GPU. The GPU is compatible with CUDA, which is used because it provides a significant speedup for PyTorch compared to running on CPU. `CUDA 10.2.89` version is used.

There are several values that are stored in memory, it is necessary to read these values out of memory so they can be used for training an AI. Addresses for elapsed song time, combo and score are needed. In order to find the addresses of these values `Cheat Engine 7.0` is used. The exact approach of retrieving the values is described later in this chapter.

4.1.2 Additional Resources for Moving

There are a few extra resources used for moving becuase a different approach is used. The python module os is used to find and load the `.osu` files of the current running beatmap from disk. Beatmapparser [Awl17] is a useful module for getting `hitObject` information out of the `.osu` files and put them in an easy to work with data structure. And lastly, Pandas is used to to save the `hitObject` information in a dataframe.

4.2 Data and Preprocessing

All these resources are used for the clicking and moving tasks to train the neural networks. The reason that the tasks are split up is because they can be trained independently. Out of experience, human players can have a good aiming skill but are bad at timing clicks or the other way around. Most importantly, it prevents interference from a non-perfect AI. For example, when training clicking and the cursor is not at the right position the click will always miss no matter how perfect the timing is. As described in chapter 3.1.5 Modifiers there are mods that replace the need for clicking or moving. To train an AI clicking the Auto Pilot mod is enabled, so no moving is needed. Similarly, in order to train moving the Relax mod is enabled, so clicking is not required.

First, the data that is used to train the AI. There is information within the game such as score, combo, and current song time that need to be retrieved from memory, this is used to evaluate the AI. Note that every few milliseconds the values are updated. As mentioned earlier, to gather the information from within the game Cheat Engine is used. Normally every time the game is launched the addresses will be different, therefore static addresses have to be found. These are retrieved by scanning memory of `osu!` for specific values multiple times. Eventually only pointers that do not change between game launches are left that eventually always point to a value like current score.
The input of the neural networks consists of four screenshots of the game window, every iteration the oldest frame is replaced by a new screenshot.

Figure 5 shows a screenshot before and after image operations. The game is running at 800×600 pixels resolution, the screenshots are taken at the same resolution with three color channels. Every screenshot is scaled down to a resolution of 80×60 and to one grayscale color channel. This reduces the amount of pixels per frame by more than 99%.

Figure 5: Left: what a human player would see when playing the game.
Right: what the same frame looks like as input for the neural networks.

4.3 Playing the Beatmaps

When learning beatmaps have to be started, to be able to learn from a wide range of different beatmaps unattended beatmaps have to be randomly selected and started. To do this, a loop is created that performs a range of actions while a beatmap is not being played. First the window title of the `osu!` process is checked, when a beatmap has been selected and play has been pressed the game window title contains the beatmap name and difficulty. This is simply checked by using the length of the title. Once the title length is confirmed to contain a beatmap name the program waits for the start of the song by reading that value out of memory. Then finally, the specific algorithm will start for the task that is being trained. Every 10th beatmap played is used for testing.

To reduce complexity of the tasks only one type of object is taken into consideration, the hit circle. Hit circles are usually the most common object in beatmaps. The beatmaps used that only consist of hit circles are created seperately. Each circle is placed somewhat randomly making sure they do not appear in the same spot on the screen. The time between hit circles is chosen in such a way that only one object is in a frame at a time, or the object is just about to be cleared when the next object starts appearing. This should decrease complexity of learning to play the game as well. The beatmaps are combined into one set that can then be randomly selected from.

4.4 Convolutional Neural Network

In the two subsections after this the approach for training of each of the tasks will be described. Before that some commonalities between the two approaches are described here. Both methods will use a Convolutional Neural Network (CNN). A number of convolutional layers and fully connected layers are used. The input is 4 low resolution grayscale frames from the gameplay, so the first convolutional layer has a depth of 4, the resolution goes down and depth increases until the network is flattened and connected to fully connected linear layers. Figure 6 shows an example network that is used for the tasks. The number of layers, layer size, filter dimensions, and depth vary in the experiments.

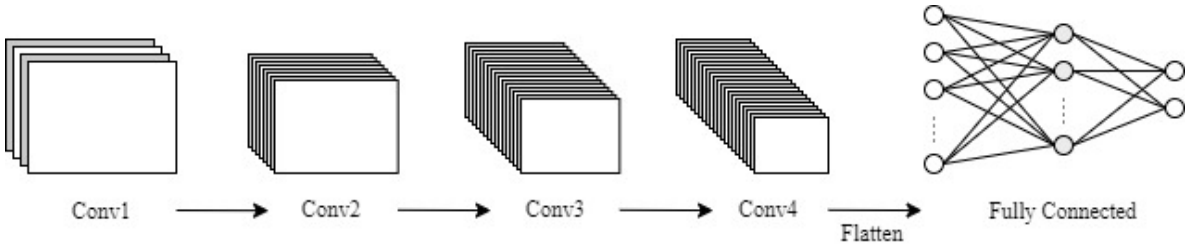

Figure 6: Example neural network diagram for the clicking and moving tasks. The depth increases, and the width and height decrease after each convolutional layer.

Since pixel data is used and every value in the convolutional layers represents one pixel value from the input, the ReLU activation function is used so that negative values after going through a layer are ignored.

4.5 Clicking Task

First, the clicking network is worked on. The reason this task is done first is because the output of this network would be fairly simple, one of two actions have to be performed: do nothing or send a click. For this task a similar algorithm to *DeepMind*'s reinforcement learning approach for Atari games is used. They propose a method they call deep Q-learning that uses a technique called experience replay[Lin92]. Q-learning methods are applied to randomly selected experiences from a set of samples stored in the replay memory[MKS+13]. This method looks promising for the clicking task because pixel data is used as input and two actions exist as output. For efficiency reasons, the replay memory is initialized when the program starts, so samples from the previous beatmap still exist in the replay memory when the next beatmap starts. A random action is taken 10% of time initially, this decreases to 0.01% after 1 000 000 iterations.

4.6 Moving Task

For moving deep Q-learning does not seem applicable, because the output will be coordinates and not a certain action. Instead, a supervised learning approach is used for this task. We know that the exact coordinates and the timestamp of objects are stored in the beatmap files. Once it is determined a beatmap has started, the file corresponding to this beatmap is parsed. A dataframe of objects is created containing the timestamp and coordinates of the object. The coordinates are used to evaluate the outputs of the network.

It has to be determined what the next object on screen is and when it is visible. It can be estimated when it is visible by using the approach rate set in the beatmap's difficulty settings. Every approach rate value has a certain time an object will be on screen before it is meant to be hit. The next object to appear on screen is the first object that has a timestamp value higher than the current timestamp of the song.

5 Experiments

Many experiments were conducted over the course of several months, not all of them can be discussed. The experiments that provides the biggest impact by resolving issues during those experiments and the ones that resulted in a satisfactory level of playing are discussed in this chapter.

5.1 Clicking Task

There are three experiments for the clicking task that gave new big insights in fixing some problems or ended up with a decent working AI.

One of the first experiments uses the deep Q-Learning algorithm that *DeepMind* used for the Atari games. A reward function with the values as shown in Table 2 was used. The action indicates what action was taken in the current iteration. Score is the minimum score earned from the action in the current iteration. Miss is yes when the combo is reset to 0. And finally, the reward could be a negative value, small positive reward or a big reward. The idea is the same as a human player, hitting objects for 300 points gains the biggest reward, 100 little less and 50 the least of the hits. Missing would mean losing combo which is discouraged as well.

Action	**Score**	**Miss**	**Reward**
Do Nothing	0	No	0.1
Do Nothing	0	Yes	-1
Click	300	No	10
Click	100	No	3
Click	50	No	1
Click	0	Yes	-1

Table 2: Variables and the reward earned for each combination in the first experiment.

In Table 3 the layers and parameters of the neural network are shown. Other than that a minibatch size of 8 is used and an initial epsilon of 0.1.

This experiment did not result in a working AI. All the results coming back from testing are 0 hits and thus 0 points total. It is not that the AI did not learn anything, during training circles were being hit all the time. It was deduced that the problem

Layer	Inputs	Outputs	Kernel	Stride
Conv1	$4 \times 60 \times 80$	$32 \times 19 \times 26$	5×5	3
Conv2	$32 \times 19 \times 26$	$64 \times 5 \times 8$	5×5	3
Conv3	$64 \times 5 \times 8$	$64 \times 3 \times 6$	3×3	1
fc4	$64 \times 3 \times 6$	128	–	–
fc5	128	2	–	–

Table 3: Layers and parameters used in the first experiment.

has to do with the time screenshots are taken. During testing screenshots are taken more often than during training, because no weights have to be updated, so the testing algorithm runs faster. To resolve this issue a framerate limiter was added so screenshots are consistent between training and testing. Every $\frac{1}{15}$th of a second a new screenshot is taken, this is the highest stable rate the system could support.

The second experiment discussed did gain some progress towards a working AI. This time the fixed framerate is used and a small adjustment to the reward function was made to reduce the complexity. The action do nothing does not give a reward now. Table 4 contains the reward values used in this experiment.

Action	Score	Reward
Do Nothing	0	0
Click	300	10
Click	100	3
Click	50	1
Click	0	-1

Table 4: Variables and the reward earned for each combination in the second experiment onward.

The layers and parameters are unchanged in the second experiment, so the same values of the first experiment are used. After approximately 96 hours of learning the AI has an accuracy between 40% and 60% at the end of a beatmap when testing.

Because it took a relative long time of learning and the performance is still poor a change to the replay memory was made. Instead of one experience replay memory, two are used, one for each action. Clicking is an action that happens relatively few number of times compared to doing nothing, so the chance of picking a sample with a do nothing

action is a lot higher. The two replay memories get combined and samples are picked from a set containing a roughly equal amount of each action.

Finally, in for the last experiment some changes were made to the experience replay and to the neural network layers that happened in earlier experiments not mentioned in this chapter. The network layers and paramaters are shown in Table 5. The same reward function as in the second mentioned experiment was used.

Layer	**Inputs**	**Outputs**	**Kernel**	**Stride**
Conv1	$4 \times 60 \times 80$	$64 \times 19 \times 26$	5×5	3
Conv2	$64 \times 19 \times 26$	$128 \times 8 \times 11$	5×5	2
Conv3	$128 \times 8 \times 11$	$128 \times 6 \times 9$	3×3	1
Conv4	$128 \times 6 \times 9$	$128 \times 4 \times 7$	3×3	1
fc4	$128 \times 4 \times 7$	512	–	–
fc5	128	2	–	–

Table 5: Layers and parameters used in the final experiment.

The graph in Figure 7 shows the accuracy over time when testing at certain number of beatmaps played. The horizontal axis indicates the number of beatmaps played, and the vertical axis the accuracy in percentages. Playing through one beatmap equals approximately 1000 iterations. The total learning time for this experiment was about 48 hours. Table 8 in Appendix A shows the complete results of this experiment containing the hits and misses at the different playcounts.

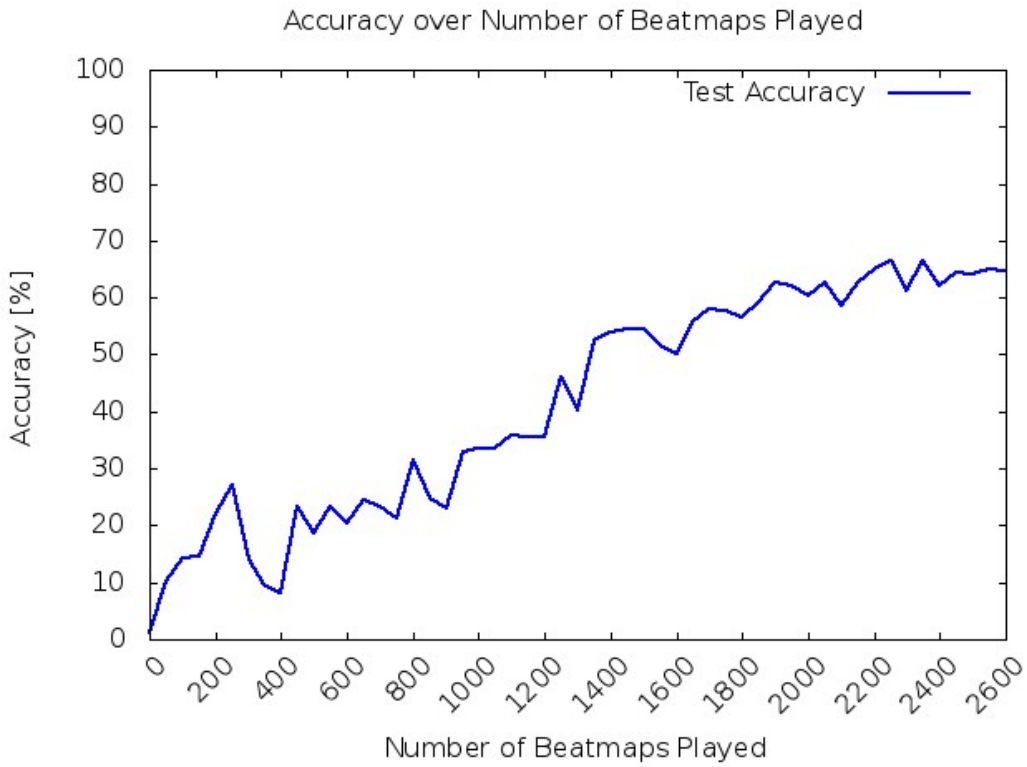

Figure 7: Accuracy over the number of beatmaps played from the final clicking experiment.

The graph shows that over time the accuracy, and therefore performance, of the AI improves with more beatmaps played. At the end of the experiment the AI gets an accuracy of around 65%, a significant performance improvement since the start.

5.2 Moving Task

The experiments for the moving task were done using supervised learning. This time training happens while using the Relax mod, so clicking is not required. The actual coordinates of the next object is determined by the current timestamp of the song, when the timestamp of previous object has passed the next object is used for evaluation of the mouse cursor position of the AI. As learned from the clicking experiments a fixed framerate is necessary, 25 fps was used. So one beatmap played is roughly 1600 iterations.

In the first experiment, and several similar ones with this setup, the output of the neural network was an $N \times M$ grid representing coordinates the cursor can move to, this grid is smaller than the size of the game window so many coordinates are unused. However, with this method the weights were not being updated. The results of all these experiments are that nothing was learned with multiple attempts running over 72 hours total. In fact, the weights at the start compared to the weights at the end show that they are exactly the same, thus the AI did not learn anything.

For the second experiment, the output of the network was altered to be two values: one for the x coordinate and one for the y coordinate. Table 6 shows the layers of the network used. In the first fully connected layer (*Fc4*) 2 additional inputs are given, the x and y coordinates of the current cursor position.

Layer	**Inputs**	**Outputs**	**Kernel**	**Stride**
Conv1	$4 \times 60 \times 80$	$64 \times 19 \times 26$	5×5	3
Conv2	$64 \times 19 \times 26$	$128 \times 8 \times 11$	5×5	2
Conv3	$128 \times 8 \times 11$	$128 \times 6 \times 9$	3×3	1
Fc4	$128 \times 6 \times 9 + 2$	2048	–	–
Fc5	2048	1024	–	–
Fc6	1024	2	–	–

Table 6: Layers and parameters used in the second moving experiment.

Unfortunately, the results of the second experiment are also all zero from testing. However, the AI did learn to move the cursor, the behaviour changes slightly over time but not the right thing. One possible explanation is that the AI cannot learn properly because when there are no objects visible the next object is still set at certain coordinates and it tries to predict the coordinates during those black screen as well.

In the third experiment a change to the learning algorithm was made, a check for when an object is on screen has been added. The weights of the network are only updated when an object is actually on the screen. The time before an object is considered to be on screen is based on the highest possible AR value, which means that the object fades

in at $300ms$ before the object is meant to be hit. Clicks are allowed to be slightly after the perfect time as well, so $100ms$ after the object's timestamp is deemed to be visible as well. From an earlier experiment, not mentioned here, the parameters have slightly changed. One fully connected layer was added (*Fc7*), and the current mouse coordinates are used in *Fc6* now. See Table 7. The network was left to learn for approximately 24 hours for this experiment.

Layer	**Inputs**	**Outputs**	**Kernel**	**Stride**
Conv1	$4 \times 60 \times 80$	$64 \times 19 \times 26$	5×5	3
Conv2	$64 \times 19 \times 26$	$128 \times 8 \times 11$	5×5	2
Conv3	$128 \times 8 \times 11$	$128 \times 6 \times 9$	3×3	1
Fc4	$128 \times 6 \times 9$	2048	–	–
Fc5	2048	512	–	–
Fc6	$512 + 2$	128	–	–
F76	128	2	–	–

Table 7: Layers and parameters used in the third moving experiment and onward.

With this experiment all results are 0 again. When observing training, the cursor now moves further towards the bottom right when training. It then creates an arc that moves over the object to be hit from the bottom right to the top left of the screen. The cursor moves over the object, but not for long enough for the Relax mod to get a hit.

The final experiment used the same network as the previous experiment, the only difference is that the time an object is considered to be on screen has been reduced significantly. From $300ms$ in the previous experiment to $60ms$ and the $100ms$ to $40ms$ for after for a total considered visible period of $100ms$. Figure 8 shows the performance in accuracy over time with beatmaps played. Table 9 in Appendix B shows all of the testing results. The low count of 100 and 50 hits is because the Relax mod always tries to time click perfectly unless it is not possible anymore.

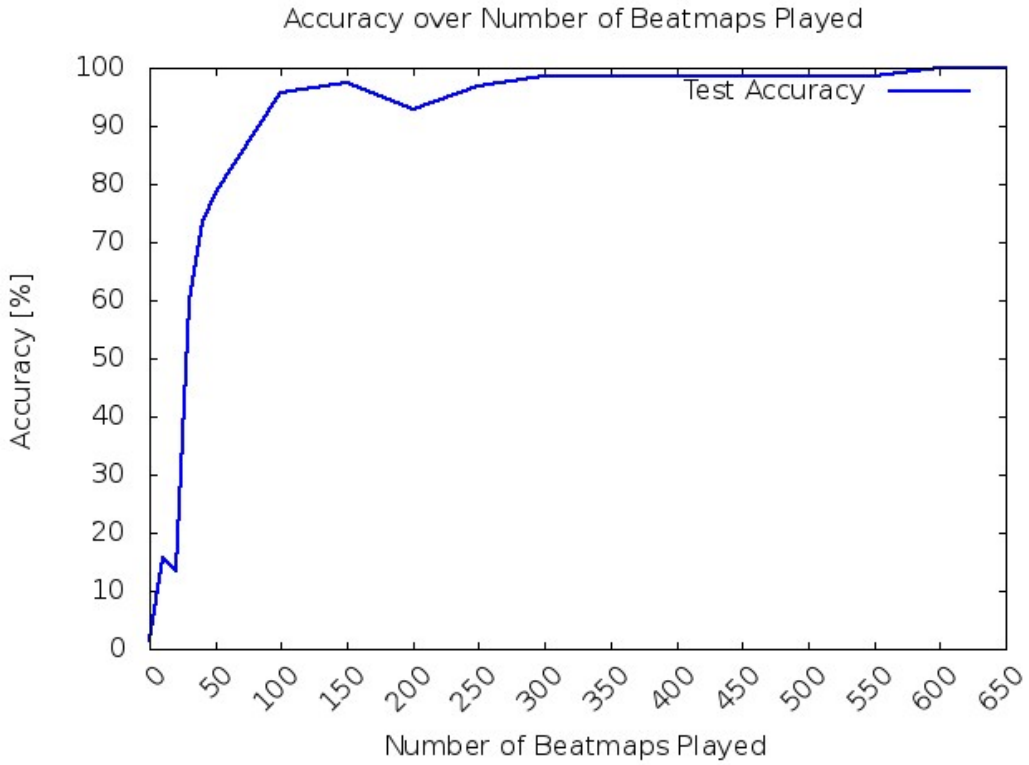

Figure 8: Accuracy over the number of beatmaps played from the final moving experiment.

The results show that using the supervised learning method and a small time window an object is deemed to be on screen, the network learns to move the mouse to the object almost perfectly after 100 beatmaps played, or around 160 000 iterations. Over the course of the other 500 beatmaps played performance becomes slightly more consistent. The observation is that when a miss does occur, it is fairly close and the cursor was hovering over the object before and after.

5.3 Discussion

In the experiments for clicking and moving many challenges had to be overcome. All the insights gained from earlier experiments that did not end up with satisfactory results helped in creating a working AI. There are many unresolved issues that exist with regards to reading data from memory. One of those issues is game patches, that happen

regularly, which can result in some of the pointer offsets for the values needed from memory to break. Every time a new update happens the process of retrieving the offsets for an address has to be repeated. Currently, however, the biggest issue is that the game will automatically close when `Cheat Engine` is open, this was most likely implemented at some point to prevent actual cheaters.

Compared to *Vedal*'s approach on YouTube. Performance of the AI presented in this thesis increases a lot faster for the moving task. The most likely explanation for that is that instead of mimicking the states of cursor and objects to be as similar as possible to the perfect playthrough, using the exact coordinates of objects is less noisy and more accurate. This might suggest that the approach used in this thesis is better suitable for these types of tasks.

6 Conclusions and Further Research

Many experiments did not yield satisfactory results, but did give some insight in the problems of the setup used and contribute to a working AI that can play the game to a reasonable level. There is a significant improvement in the timing of clicks over the learning period using a reinforcement learning algorithm. However, the time it takes to train the AI to the current level of 65% is rather long. There is room for improvement for the clicking task so that an acceptable accuracy of 95%+ from a player perspective can be reached. The moving task using a supervised learning method showed that the AI learned to hit almost all objects and reaches a high accuracy. It did not require a lot of time, with only a play count of about 100 songs to reach this level. Overall it has to be concluded that it is possible to create an AI for `osu!`. The most difficult problems were tackled, but other difficult challenges remain. It is therefore not easy to create an AI for `osu!` with the current applied methods.

Perhaps after improvements the tasks the AI learned could be used in robots. Determining targets to move to, then moving the cursor to the location of the target could be altered to move a physical objects to certain targets. The clicking tasks relates to timing, if holding clicks is implemented also, the two tasks combined could be used to pick up targets and move them to different locations. These could be topics for later research.

To further expand upon this topic a lot of further research can be done within the game itself. Most importantly, sliders and spinners have to be included to be fully able to play the game, this will require many adjustments to the current algorithms. Clicking has to be split up in keydown and keyup events to be able to clear sliders and spinners, because they require a keypress to be held down. Spinners only have coordinates for the center of the screen, but the player requires to circle around this center, some method has to be figured out so the AI can learn to do that. Next, there are many improvements to be made in many aspects to optimize the learning time and performance. One important performance aspect is regarding to the framerate, a low framerate is suboptimal because clicks and movements can only be made at those intervals. Timing is a very important aspect of the game, only being able to click once every 60ms could impact the speed of learning and the overall performance as perfect hits are less likely.

References

[Awl17] Awlexus. Python osu parser. https://github.com/Awlexus/python-osu-parser, 2017. [Accessed 30 July 2020].

[faqnd] Faq knowledge base — osu! https://osu.ppy.sh/help/wiki/FAQ, n.d. [Accessed 5 June 2020].

[Her07] D. Herbert. welcome — osu! https://osu.ppy.sh, 2007. [Accessed 5 June 2020].

[Her16] D. Herbert. ppy/osu: rhythm is just a *click* away! https://github.com/ppy/osu, 2016. [Accessed 5 June 2020].

[Lin92] Long-Ji Lin. *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, USA, 1992.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. 2013.

[Pi8] N. Pičuljan. Pytorch reinforcement learning: Teaching ai how to play flappy bird. https://www.toptal.com/deep-learning/pytorch-reinforcement-learning-tutorial, 2018. [Accessed 10 March 2020].

[Set06] SethBling. Sethbling - youtube. https://www.youtube.com/c/MinecraftSethBling, 2006. [Accessed 2 August 2020].

[Ved14] Vedal. Vedal - youtube. https://www.youtube.com/user/Vedal987, 2014. [Accessed 2 August 2020].

A Final Clicking Experiment Results Table

Table 8: Results of the final clicking experiment. The accuracy, number hits of different categories, and the number of misses are shown.

Playcount	**Accuracy [%]**	**300 hits**	**100 hits**	**50 hits**	**Misses**
0	1.30	0	2	1	61
50	10.16	2	8	11	43
100	14.32	2	17	9	36
150	14.58	2	11	22	29
200	22.14	6	15	19	24
250	27.08	7	19	24	14
300	14.06	1	9	30	24
350	9.64	0	4	29	31
400	8.07	0	3	25	36
450	23.18	3	21	29	11
500	18.75	1	16	34	13
550	23.44	1	27	30	6
600	20.31	0	28	22	14
650	24.48	2	28	26	8
700	23.44	2	25	28	9
750	21.35	0	22	38	4
800	31.51	6	34	17	7
850	24.74	1	28	33	2
900	22.92	2	21	34	7
950	33.07	7	32	21	4
1000	33.59	8	32	17	7
1050	33.59	7	34	19	4
1100	35.94	9	30	24	1
1150	35.68	8	33	23	0

Table 8 continued from previous page

Playcount	Accuracy [%]	300 hits	100 hits	50 hits	Misses
1200	35.68	8	33	23	0
1250	46.09	16	34	13	1
1300	40.36	12	31	21	0
1350	52.60	22	29	12	1
1400	53.91	23	29	11	1
1450	54.43	23	32	7	2
1500	54.43	26	23	7	8
1550	51.56	22	27	12	3
1600	50.00	22	25	10	7
1650	55.73	26	26	6	6
1700	58.07	30	18	7	9
1750	57.81	30	19	4	11
1800	56.51	26	27	7	4
1850	59.11	29	25	3	7
1900	62.76	33	19	5	7
1950	61.98	33	18	4	9
2000	60.42	32	19	2	11
2050	62.76	33	20	3	8
2100	58.59	28	25	7	4
2150	62.76	33	20	3	8
2200	65.10	33	24	4	3
2250	66.41	35	22	1	6
2300	61.20	30	26	3	5
2350	66.41	34	25	1	4
2400	62.24	31	24	5	4
2450	64.32	34	21	1	8
2500	64.06	32	24	6	2
2550	65.10	33	24	4	3
2600	64.84	32	27	3	2

B Final Moving Experiment Results Table

Table 9: Results of the final moving experiment. The accuracy, number hits of different categories, and the number of misses are shown.

Playcount	**Accuracy [%]**	**300 hits**	**100 hits**	**50 hits**	**Misses**
0	1.56	1	0	0	63
10	15.63	10	0	0	54
20	13.54	8	1	2	53
30	59.64	38	0	1	25
40	73.44	47	0	0	17
50	78.65	50	1	0	13
100	95.83	61	1	0	2
150	97.40	62	1	0	1
200	92.97	59	1	1	3
250	96.88	62	0	0	2
300	98.44	63	0	0	1
350	98.44	63	0	0	1
400	98.44	63	0	0	1
450	98.44	63	0	0	1
500	98.44	63	0	0	1
550	98.44	63	0	0	1
600	100.00	64	0	0	0
650	100.00	64	0	0	0