Master Computer Science

Creating an AI that plays Suika game

Name:	Julian Poelsma
Student ID:	S2225387
Date:	6/5/2025
Specialisation:	AI
1st supervisor:	Mike Preuss
2nd supervisor:	Matthias Müller-Brockhausen

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

1 Abstract

Suika game is a physics based Japanese video game. Players drop fruit into a box and merge identical pieces of fruit into larger ones. The goal of the game is to strategically combine fruits, such that the box does not overflow. These pieces of fruit are affected by gravity and can collide with other pieces. The game ends when the box overflows and no more fruits can be added.
In this thesis, a custom version of Suika game, which allows algorithms to play automatically, is created. After benchmarking these algorithms against each other and against a human player, the best algorithm was found to be Monte Carlo based. By exploring a small sample of the possible actions and simulating what happens after playing these actions, Monte Carlo based approaches were able to comfortably select actions that beat other machine learning approaches, such as reinforcement learning. Monte Carlo even managed to score better than the human player on average.

Contents

2 Introduction

With the rise of AI, the intersection between artificial intelligence and gaming is very interesting to research. In recent years, more and more AIs have been created that are able to play games and outperform the best human players. Googles AlphaZero has been able to outperform the worlds best human players in games such as Chess and Go (Silver et al. 2018). This not only demonstrates that AI can play these games intelligently, it was also able to figure out the best strategy on its own and learn from previous experiences.

In this research project, an AI will be created that plays Suika game, a physics based puzzle game. The puzzle elements of the game combined with the physics and randomness make this game very interesting to play and to research on. Since no two turns are the same, playing this game requires dynamic decision making and adaptability.

The aim of this project is to answer the research question: Can an AI be created that plays Suika game?
To answer this research question, a custom version of Suika game is made that allows computer players to play the game themselves. With this version, computer players will be developed to play the game as well as possible. These computer players will use different AI methods and game strategies to find the best scoring method, which will be discussed in later sections.

3 Background

Suika game is a simple game that is developed by a Japanese company called Aladdin X. It originally launched on the Nintendo Switch in 2021, but currently it can be played on almost any device and even in web browsers (Takahara 2023).

3.1 Gameplay

The gameplay of Suika game is often stated as a combination between Tetris and 2048. The player drops pieces of fruit from the top of the screen, which fall down due to gravity. When two identical pieces of fruit touch, they combine into a larger fruit. The aim of the game is to keep combining pieces of fruit such that the player eventually reaches the largest piece of fruit. A screenshot of the game can be found in figure 1.

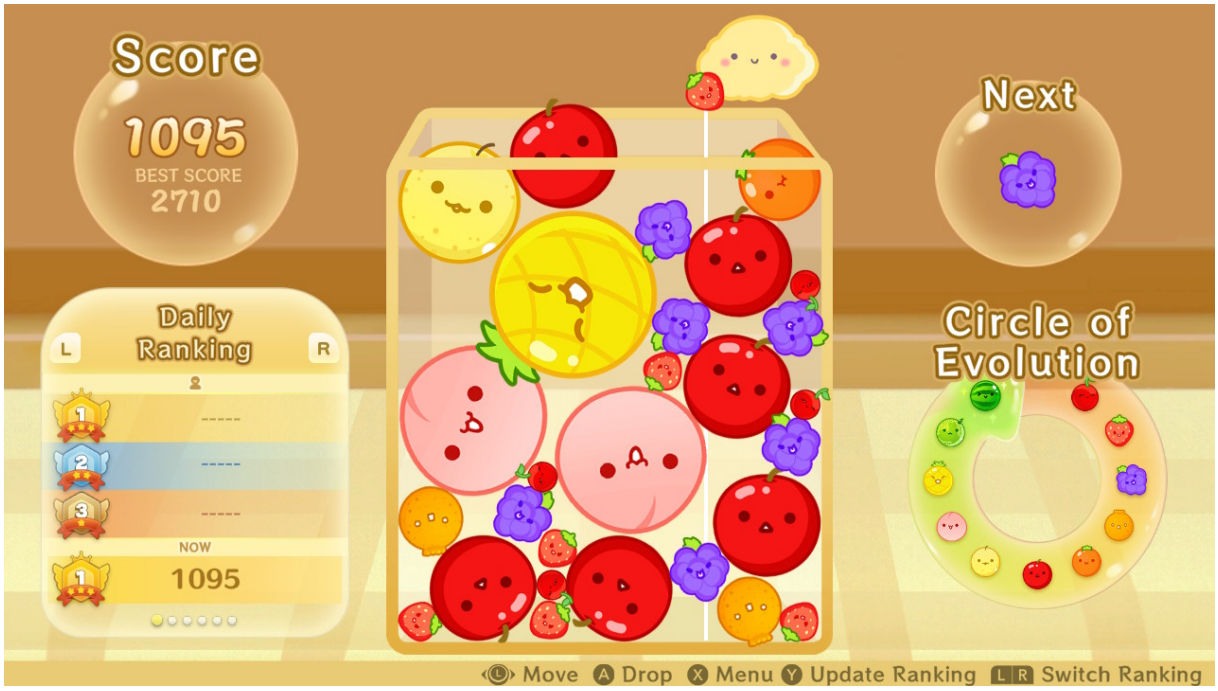

Figure 1: Original Suika game while playing

The pieces of fruit are dropped in a two-dimensional box. Each dropped fruit raises the height that the fruit reaches, but a successful combination can decrease the height. The game finishes when the box is filled, the max height is reached and a piece of fruit goes outside the box. This means that while playing the game, the player needs to drop fruit that combines into larger fruit, while also accounting for the space that that fruit takes up. Ideally, there would be as little duplicates of fruit types as possible, since each extra duplicate takes up space.

The piece of fruit that the player can drop is randomly selected from the smallest five fruit-sizes. The player can see which fruit is currently able to drop and which fruit will be the next fruit that will drop. This allows for some strategic planning and limited forward thinking.

3.2 Strategy

In order to reach the highest score, it is important that the fruits are placed as efficiently as possible. The main aim of the game is to combine fruits into larger fruits, so it is a good strategy to try to always combine the dropped fruit with a fruit that already existed. While doing this, the player must ensure that space is not wasted and that larger fruits can also easily combine.

In picture 1, a screenshot of a game that is not played optimally can be seen. Whilst there are some larger fruits, it is almost impossible to combine them, since there are smaller fruits in-between. In this case, especially the purple grapes and red strawberries prevent the red apples from touching each other. If instead these red apples were able to touch each other and merge, the playing field would be much smaller, and the score would be much higher. It can also be seen that there are many duplicates of other fruits of the same size. All of these duplicates take up space and would ideally be combined into larger fruits.

To combat this, the player can try to place fruits of subsequent types next to each other by ordering the fruits from small to large. This allows the user to eventually drop one fruit on the smallest fruit, which will start a chain reaction and combine many fruits in a turn. In picture 2, you can see such a setup. By dropping a single purple grape in a single turn, it is possible to combine the two blue purple grapes into a a light-orange dekopon, which will then combine with the other dekopon into a orange persimmon. These two orange persimmon will then also combine. This process goes on for two more steps until eventually the playing field is nearly empty and the light yellow pear is converted into a pink peach.

Aligning the playing field like this is very tricky to do, but gives the opportunity to create large chains and combine many pieces of fruit at once.

Figure 2: Playing field with opportunity to combine multiple circles at once. Screenshot of original Suika game.

Using another strategy, it is possible for the player to move circles that have already been placed. This can be achieved by placing new circles on a specific side of an existing circle. If the weight that lies on the left side of the circle is higher than the weight that lies on the circle from the right side, the circle will move to the right. This can allow the player to shift the playing field and give themselves more opportunities (ktrZetto 2023).

3.3 Own implementation

In order to be able to have a computer automatically play the game, the game needs to be recreated. This was done using Python and the graphics have been implemented with PyGame. Although it is graphically simpler, the game is largely the same. There are some important

differences, which will be highlighted below.

Instead of different types of fruit, this version works with different colored circles. In total there are 15 types of circles, with each one being slightly larger than the last one. It is theoretically possible to reach the largest circle, but this is practically undoable, since the game field will fill up with circles before the largest circle is reached. This ensures that the game does not finish, but instead it gives a score

Another difference is that in the original Suika game, the game would end if a piece of fruit that was previously inside of the box bounces out of it. This mechanic has caused quite some game-overs that felt unfair. Therefore, this game ending condition has been removed in this version. Instead, the game will only go game-over if a new circle hits another circle before it enters the box. This ensures that the game only ends when the box is fully filled.

The performance of a player is evaluated by the score. The score increases when circles are combined successfully. The larger the circles that are combined, the more the score will increase. In order to evaluate different types of players, the score at the end of the game serves as the primary metric for evaluation.

Every time that two circles of the same type collide and combine, the total score is increased following the formula:

$$\text{score} \mathrel{+}= c_t * 2$$

where c_t is the number that corresponds to the type of circle that was combined. c_t starts at 1 for the smallest circle and is increased by 1 per circle size up to a maximum of 15. For example, combining two of the smallest blue circles gives a red circle and a score increase of 2. Combining that red circle with another red circle gives a score increase of 4 and a total score of 6.

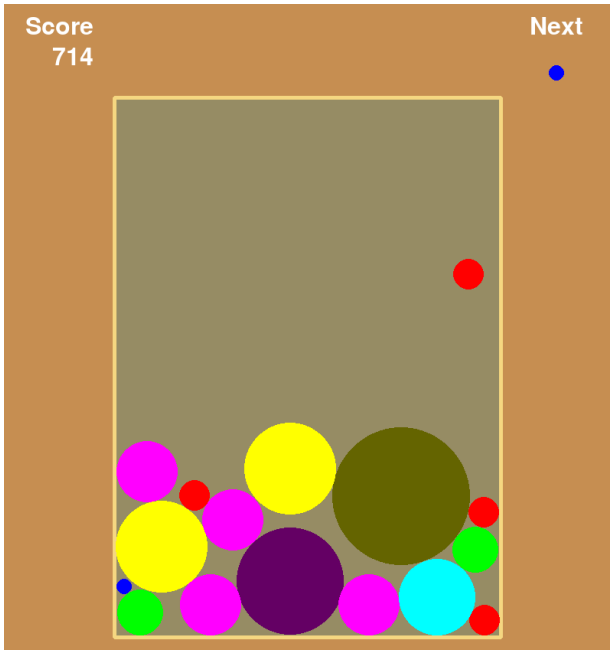

Figure 3: My version of Suika game while playing

When an AI plays my version of Suika game, a short game that does not perform that well can be expected to take around 350 turns before going game over. A normal game with a good player will take around 1000 turns before going game over and reach a score of 6500-7000. The best games, with scores over 10.000 can last more than 1500 turns.

3.4 Related problems

In order to create an AI that can play Suika game on its own, it is good to know how AI's for other similar games have been made. Tetris is a famous game and just like Suika, it also relies on the player dropping items from the top into the correct position. In the paper titled "The game of Tetris in Machine Learning" (Şimşek 2019), different approaches to having a computer play Tetris automatically are described. The paper shows that for a long time, the best algorithm to beat Tetris was an handcrafted algorithm. Currently, the best algorithm for Tetris consists of a reinforcement algorithm that has been optimized using an evolutionary algorithm in the policy evaluation step. It is currently not possible to extract the best features from a raw Tetris input. The transformation of the Tetris playing field to a list of useful features is still done by hand. No algorithm has managed to play properly with just raw inputs.

A key difference between Tetris and Suika is the number of inputs. Tetris is usually played on a field that is 10 pixels wide. Accounting for the different Tetriminoes and the different rotations that they can have, the median number of possible placements for a Tetrimino is 17 (Şimşek 2019). In comparison, the Suika playing field is 500 pixels wide, which means that there are 500 possible placements for each circle.

Another game that is similar to Suika game is the game 2048. In this game, the aim is to combine the numbers on a 4x4 tiled grid until the total number has reached 2048. The player starts with the numbers 2 and 4. A number can only be combined with the same type, which will result in a tile with the addition of these two numbers. For example, combining a 2 and a

2 will result in a tile with a 4 on it. The player can play the game by swiping in one of the four cardinal directions. This moves every tile to this side and combines the numbers. Each tile will move to the side until it reaches the edge or it reaches another tile (Rodgers and Levine 2014).

Just like Suika, 2048 relies on the player making good combinations to progress. If the player fails to combine tiles, the playing field will quickly fill up. Similarly to Suika, the best 2048 strategy is to place subsequent tiles next to each other such that a chain reaction can be made. This allows the player to easily merge tiles and keep tiles from blocking other tiles.

In the paper "An investigation into 2048 AI strategies" (Rodgers and Levine 2014), Monte Carlo Tree-Search (MCTS) and Averaged Depth Limited Search (ADLS) are applied to 2048 and the results are compared. It finds that due to the small complexity of 2048, it is relatively easy to simulate the game up to a depth of 8. The main problem with simulating the game is the randomness of the game. A good AI must minimize the risk it takes, but not too much.

4 Related work

Numerous methods and algorithms have been used in the past to enable automatic gameplay. Reinforcement learning involves an agent that trains itself using trail and error. By playing the same game over and over again, the agent can learn which action to take for a specific game state. Shao et al. 2019

Learning is done by evaluating each action taken by the agent. If the action results in a high reward, the agent will learn to use that action more often in that scenario. However, if it results in a poor reward, the agent will learn to stop using that action in that scenario. This means that every time the agent needs to make a decision, it analyzes the current game state and decides an action based on that game state. By then playing this action the game will transition into a new game state (Kaelbling, Littman, and Moore 1996).

Deep reinforcement learning is a variant of regular reinforcement learning that combines reinforcement learning with deep learning. In deep reinforcement learning, a neural network is used to transpose the game state into useful features. This allows for way larger inputs and game states compared to normal reinforcement learning. Therefore, deep reinforcement learning is commonly used when working with visual data, such as pictures (Shao et al. 2019).

Another algorithm that can automatically play games is Q-learning (Watkins and Dayan 1992). It uses a Q table, which is a state-action table that stores the expected reward of a specific action taken in a specific state. This value is called the Q-value. Upon interacting with the game, the algorithm automatically updates the table based on the rewards received. The algorithm will then select the action that maximizes the expected reward for the current state.

Deep Q-learning is a variant of Q-learning (Mnih, Kavukcuoglu, Silver, Rusu, et al. 2015), but instead of a Q table, it employs a deep neural network to predict the Q-values of state action pairs. The action that maximizes the expected reward for the current state is then selected. Predicting the rewards instead of storing them enables the algorithm to work with large environments, with a large number of states and corresponding actions.

Another algorithm capable of automatic game play is Monte Carlo. With Monte Carlo, simulation is used to find the best action to take. By repeatedly sampling from the actions that can be made, the algorithm can eventually decide which action to take, by looking at which action gave the best score. (Kroese et al. 2014)

The strength of Monte Carlo lies in the fact that it accurately simulates what is going to happen, instead of having to predict it. This allows for way more accuracy at the expense of larger computational costs due to the simulation. Monte Carlo is most powerful in cases where simulation is easy and a lot of simulations can be done (Kroese et al. 2014).

Monte Carlo Tree Search is an expansion on the original idea of Monte Carlo. It aims to build up a tree where the nodes are different game states and the vertices are the actions. From a certain state of the game, it is then possible to optimally play the game by traversing the tree and choosing the best actions (Browne et al. 2012).

5 Methods and results

Multiple approaches have been used to play the game. This section will dive into the different approaches and how they work. The results of each approach will also be shown, which allows for the comparison between approaches.

Each of these approaches returns a horizontal coordinate that indicates where the next circle should be dropped. Since the playing field is 500 pixels wide, 500 different locations can be chosen.

In order to interpret the score, multiple rounds played with the same algorithm will be considered. The median, the 25th and the 75th percentile will then be evaluated. Where applicable, behavior and tactics will also be discussed. The 25th and 75th percentiles are used because of the skewed distribution of the scores. In contrast to a normal distribution where the results are evenly spread across the mean, the scores of the different algorithms have the tendency to be skewed towards the left. Therefore, by reporting on the 25th and 75th percentile, more information is given about the variety of results. Furthermore, due to the large variance in scores by the same algorithm, which is caused by the randomness of the game, outliers are likely in the data. The 25th and 75th percentiles are less likely to change much based upon an outlier.

The results are based on 30 runs for all nonhuman algorithms. 10 runs were performed by human players.

5.1 Random player

The first player that was implemented was a random player. This was not meant to be a serious attempt, but rather a poor performing benchmark. This algorithm is not expected to perform well, but it gives a perspective to the scores obtained by other algorithms. This random player

randomly chooses a location and drops the circle there. It does not take the current state into account at all.

Median	25th percentile	75th percentile
2259	2027	2568

Table 1: Scores of the random player

5.2 Human player

In order to properly evaluate the algorithms, they need to be compared to a human player. This human player is experienced in the original Suika game and this version. The player manages to beat the original Suika game in about 20 percent of the games that are played and is familiar with the strategies of the game. This classifies the player as above average, but not the best of the world.

To generate the results shown in table 2, the human player played 10 games. With these scores, the human player managed to beat most of the algorithms, but was ultimately beaten by Monte Carlo based algorithms.

Median	25th percentile	75th percentile
5076	4824	6241

Table 2: Scores of a human player

5.3 Monte Carlo simulation

The first real approach that was tried to solve the game was Monte Carlo Simulation. With this simulation, every time that a move can be played, a number of simulations will be done. Based on the score that is achieved during simulation, the best move will be chosen and will be played in the real game.

Monte Carlo Simulation is computationally expensive, since the game needs to be simulated. Therefore it is not feasible to test every possible action for every state. For every action that is simulated, the circle is dropped and the next three seconds are simulated. This ensures that the simulation accurately shows what would happen if this move were to be played.

In cases where circles are still moving, it can be a good option to wait and see what the movement does. By simulating three seconds of game-time, it is checked if circles will combine automatically. If circles are still moving and some circles will combine, a move will not be played. Instead, the circles will get the opportunity to combine.

Monte Carlo is performed by simulating multiple different actions for a given game state. Since this is only done for a depth of up to two, no full tree will be built. Therefore, Monte Carlo is used instead of Monte Carlo Tree Search.

5.3.1 Evaluation function

After a turn has been simulated, the different possible actions and the states caused by these actions must be evaluated. This evaluation gives each state a reward and the state with the highest reward will be played.

The reward is calculated in the following way:

1. Start with a reward of 0.
2. Add three points to the reward for every circle that has been removed in this turn.
3. Add one point if the height of the highest circle has not increased.
4. Add one point for every circle that touches the circle that has just been dropped and is of the next type.

Following these criteria, the best turn consists of a turn in which as many circles as possible are removed, the total height has not increased, and the circle that was just dropped lies in a strategic position.

5.3.2 Approaches

Four different approaches are used to select the actions corresponding to a state that need to be checked. The first approach is to generate random numbers between 0 and 500. These will be used as drop-locations that will be simulated.

The second approach is step based. The aim of this approach is to divide the playing field into fields of equal sizes, each of these fields is then simulated. In practice, with 50 simulations, this means that each step has a size of ten pixels. Therefore every tenth pixel will be simulated. Unlike the random approach, this approach ensures that the entire field is covered, which cannot be ensured with the random approach.

The third approach uses type matching to select possible drop locations. It checks all circles to see which circles have the same type as the circle that is about to be dropped and can be reached from the top. For every circle that adheres to these criteria, the location of that circle will be used as possible drop-location and simulated. This is the only approach that does not do 50 simulations, since the number of simulations is depended upon the number of circles that adhere to the criteria. Due to the size of the circles, the number of circles will never be more than 50.

The fourth and final approach is a combination of random and type matching. This approach starts the same as type matching by creating a list of possible drop locations based upon the circles that are already in the field. Unlike type matching, if the list of possible drop-locations is less then 50, it will be filled up with random locations until 50 drop-locations have been selected.

How many turns are simulated for every simulation can also change. Two different approaches are distinguished here. For the first approach, one turn will be simulated for every drop-location

using only the circle that is about to be dropped.

For the second approach, two turns are simulated for every drop-location. There, the circle that is about to be dropped and the circle thereafter, which is known, will also be dropped in the same simulation. This is more computationally expensive than simulating one turn, but it also allows the algorithm to look further into the future. Due to the computational power that is required, only one drop-location per simulation is used for the second circle.

5.3.3 Simulating

For each of the simulations, the current state of the game is saved. The circle will be placed at the spot that was chosen beforehand and the game will be simulated for three seconds. This time ensures that the circle has fully reached the bottom or another circle and has mostly stopped moving. If the simulation consists of one turn, the simulation is done. The score is be evaluated and the previous state of the game is reset. If two turns are tried, the second turn will now follow in the same way as the first turn. For two turns, the score is added together and then evaluated.

5.3.4 One turn per simulation

One turn per simulation is done with 50 simulations per turn.

Model	Median	25th percentile	75th percentile
Random	6603	5496	7238
Steps	6788	5346	7387
Type matching	3444	3140	4280
Combined	6234	5522	6838

Table 3: Monte Carlo one turn per simulation

5.3.5 Two turns per simulation

Two turns per simulation are done with 50 simulations per turn. This takes twice the computational power as one turn per simulation with 50 simulations per turn. Due to the poor results, type matching is not included here.

Model	Median	25th percentile	75th percentile
Random	6947	6033	8209
Steps	6385	5806	7833
Combined	7353	5946	8227

Table 4: Monte Carlo two turns per simulation

Two turns per simulation are done with 25 simulations per turn. This takes the same computational power as one turn per simulation with 50 simulations per turn. Due to the poor

results, type matching is not included here.

Model	Median	25th percentile	75th percentile
Random	6672	6180	7160
Steps	6676	5585	7798
Combined	5789	5364	7157

Table 5: Monte Carlo two turns per simulation with 25 simulations per turn

5.3.6 Behavior

Monte Carlo knows what will happen in the turn that it plays. Therefore the best move is not simply a prediction. This can be seen when looking at the game when playing. When Monte Carlo is used, it can be seen that the games physics are used really well. For example, sometimes balls are nudged against each other in order to cause them to merge, before the ball bounces to the other side of the box and merges with a ball there. Because of this, when looking at the game, it looks like the player is doing trick-shots all the time.

5.4 Handcrafted Type matching

This handcrafted algorithm aims to drop a circle on another circle of the same type that is already laying in the field. From a collection of all circles that are visible from the top and of the same type as the circle that has to be dropped, a random circle is chosen. The circle will then be dropped on that circle. If no circle of the same type can be found, a random location will be chosen.

This ensures that if the circle can be matched with another circle of the same type, it will be.

Median	25th percentile	75th percentile
3467	3078	4012

Table 6: Scores of type matching algorithm

5.5 Deep Q learning models

The first machine learning model that was tried was a Deep Q learning model. It is a classification model where the output is a 500 long array of floats, which corresponds with all the possible drop locations (Mnih, Kavukcuoglu, Silver, Graves, et al. 2013). The input will be explained below.

The final model consists of three layers. The first layer is the input and the third layer the output, while the second layer uses a relu activation function. To train the model, the AdamW optimizer from PyTorch is used alongside the L1Loss function to calculate mean absolute error and optimize based upon these results (Loshchilov and Hutter 2019)(Janocha and Czarnecki

2017). Other models have been tried as trainer, but these gave the best results.

For the evaluation function, the same criteria are used as in Monte Carlo. This means that points are earned for every combination that is made, if the max height does not increase and if the circle touches other strategic circles.

Each model is trained by playing a 1000 games until game over. After this training, the model is saved and used to generate scores for evaluation. For the evaluation, 30 games are played per model.

5.5.1 Normal topview

The input of this model was the view of the top, which means that for every pixel, the input would show which circle lay there with a number. Along with this, the circle that is about to be dropped and the circle that is next will also be given as input. It uses a classification algorithm and the output thus consist of 500 nodes, each corresponding with a pixel to drop the circle.

5.5.2 Simple topview

The simple topview tries to make the training easier by making the input simpler. Instead of showing every circle in the game, only the circle of the same type as the circle that is about to be dropped are shown. This means that every input is 0, except for the places where a matching circle is found.

In order to decrease the input size, the input can be sliced into blocks of equal size. When this is done, the most occurring circle in a block will be shown to the model as input. Hyperparameter optimization was used to find the optimal hyperparameters. Since the rewards that are given already accounts for future rewards, the gamma will be kept relatively high. The best hyperparameters for normal topview were found to be a learning rate of 0.01, 50 slices and a gamma of 0.99. For the simple topview, a learning rate of 0.1, 50 slices with 10 pixels per slice and a gamma of 0.99 were found to give the best result. With these parameters, the following scores have been achieved:

Model	Median	25th percentile	75th percentile
Normal topview	2903	2122	3407
Simple topview	2692	2062	2846

Table 7: Deep Q Learning results comparing simple and normal topview.

Learning rate	Slices	Gamma	Median	25th percentile	75th percentile
0.1	50	0.99	2436	1858	2980
0.1	500	0.99	2740	2416	2970
0.05	500	0.99	2231	1495	2674
0.01	500	0.99	2482	2194	2782
0.05	50	0.99	2617	1887	3270
0.01	50	0.99	2903	2122	3407
0.01	50	1	2432	2304	2646
0.01	50	0.95	2490	2127	2641
0.01	25	0.99	2466	1923	2941
0.1	25	0.99	2237	2074	2554
0.1	100	0.99	2376	2222	2754
0.01	100	0.99	2706	2436	2930

Table 8: Results of different hyperparameters with normal topview model.

Learning rate	Slices	Gamma	Median	25th percentile	75th percentile
0.01	50	0.95	2387	2041	2736
0.01	50	0.1	2414	2164	2806
0.01	50	0.99	2414	2164	2806
0.05	500	0.99	2516	2268	2732
0.1	50	0.99	2692	2062	2846
0.1	500	0.95	2470	2274	2754
0.01	50	1	2432	2304	2646
0.01	500	0.95	2426	2058	2976
0.05	500	0.99	2580	2086	2906
0.05	50	0.99	2604	2168	2906
0.1	50	0.5	2514	2240	2886

Table 9: Results of different hyperparameters with simple topview model.

5.5.3 Behavior

When watching the models play, it can be seen that most of the circles are dropped close to each other, but not necessarily combined with circles of the same type. Both deep Q learning models have the tendency to drop all circles on the same spot. This occurs when a spot gives a high reward. That location will then be repeated and if it manages to merge another circle, the chance of that location being chosen again only rises. This strategy can deliver some quick points, but this can also quickly raise the height of the circles and is generally not a very good strategy.

5.6 Linear Regression

Linear Regression is also used to select the correct drop location. For this approach Linear-Regression from sklearn is used.

Since the Deep Q learning model did not properly train, another approach was tried. This approach aims to decrease the amount of input and outputs which should make the problem

easier to learn.

The input for this model is based on a specific drop-location. Instead of being able to see the whole field at once, the model only sees a few details based on a specific location. The input for the model consists of five values which can either be 0 or 1. The first value is based upon what is directly underneath the current drop-location. This will be 1 if and only if the circle directly beneath it is of the same type as the circle that is about to be dropped. The second and third value look to the left and right respectively. These values will be 1 if and only if a circle of the same type is found to the left or right of the circle directly beneath the drop-location. The fourth and fifth value are depended upon the rest of the playing field and also look to the left and right respectively. These values will be 1 if and only if a circle of the same type as the circle that is about to be dropped can be found left or right of the current drop-location. This includes the whole field, which allows it to basically see if a circle of the same type exists in the playing field.

The output is a classification output with three variables. The one with the highest value is then used to select an action. These outputs correspond to the actions of moving the drop-location a pixel to the left or to the right, or dropping the circle at the current location.

The combination of this allows the drop-location to move across the playing field as a cursor, constantly seeing what lies beneath it and deciding the next action based upon that.

During training of the model, the evaluation of previous moves is done in a simpler way than in the models discussed above. The number of points in the reward are directly correlated with the amount of circles that are removed during the turn and can be calculated by subtracting the score before the turn from the score after the turn. This means that the number of points in the reward is simply the amount of points that was earned in that turn.

In order to help the model with training, this model has also been tested with user-created data. To do this, 5000 inputs have been randomly generated. Based upon these inputs, the corresponding outputs are generated. This data is used to train the model before playing.

User-created data	Median	25th percentile	75th percentile
Yes	3974	3232	4568
No	2108	1934	2599

Table 10: Linear regression results

5.6.1 Behavior

When looking at the model that plays, it can be seen that the model that is pretrained using user-created data is very good at type matching. It is able to find a circle of the same type and drop the circle at that location. The model that is not pretrained does not match that well, which shows that pretraining helps the model learn the problem.

5.7 Q Learning

The last machine learning model that was used was Q learning (Watkins and Dayan 1992). Since the linear regression model was able to work with the smaller input and output, This model tries to learn the states of the game and the corresponding actions that will amount to the highest score.

The best hyperparameters were found using hyperparameter optimization. The best hyperparameters were found to be a learning rate of 0.2 and an epsilon value that starts at 0.5, but slowly decays until it has reached 0.1. This ensures that there is a lot of exploration at the beginning, but when more turns are played and more has been learned, the amount of exploration decays and more exploitation is done. Q learning uses a Q table to keep track of the states and their score. This table is 2^5 by 3, since the input consists of 5 values that can all be either zero or one, and the output consists of one of three possibilities.

The input, output and evaluation function that are used are the same as for Linear regression.

Median	25th percentile	75th percentile
2741	2529	2806

Table 11: Q Learning Results with learning rate of 0.2 and epsilon between 0 and 0.1

Learning rate	Median	25th percentile	75th percentile
0.01	2230	1938	2720
0.05	2452	1908	2726
0.08	2741	2529	2806
0.1	2528	2166	2770
0.15	2560	2396	3011
0.2	2574	2077	2746

Table 12: Q Learning Results with changing learning rate

5.8 Combined results

Figure 4 shows the best results of every model and compares them to each other.

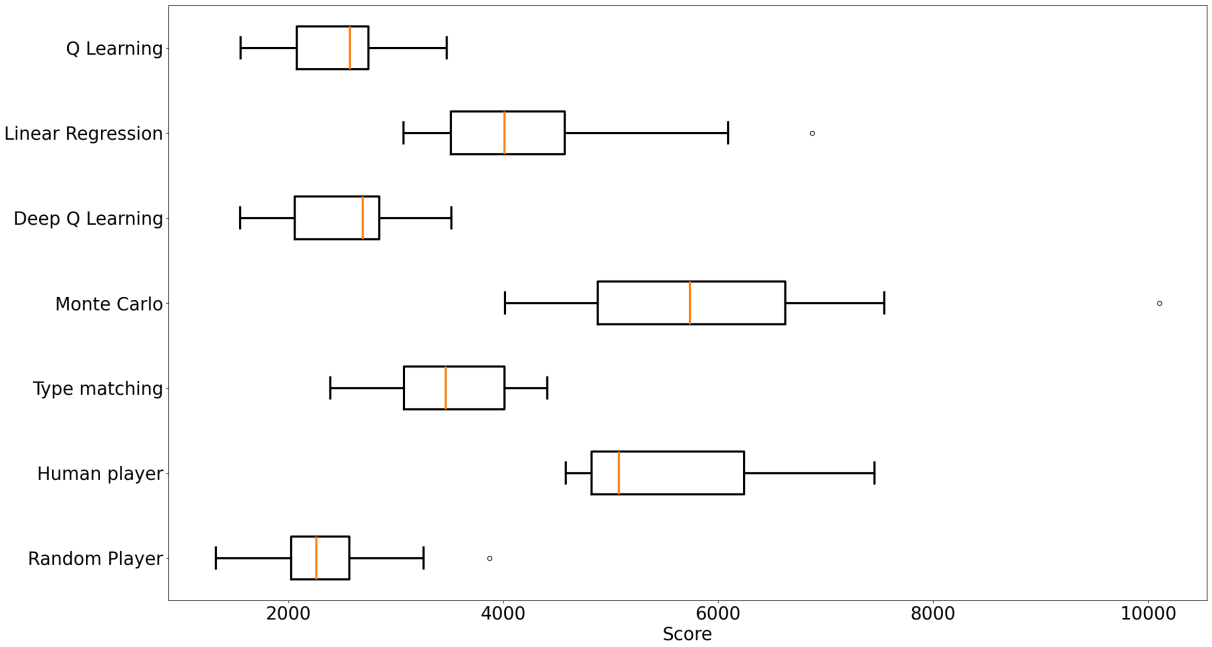

Figure 4: Boxplot with scores per model. The orange line is the median and the box surrounding it shows the 25th and 75th percentile. The line extends until the 0th and 100th percentile, with the circles showing outliers. 30 runs were performed for every non-human algorithm. The human player did 10 runs.

6 Conclusion and discussion

When looking at the results, it can be seen that the Monte Carlo approaches worked the best and achieved the highest score. After that the human player performed quite well. The different machine learning approaches worked less good, depending on how well training went.

Based on this, the different approaches can be divided into three different groups. The lowest scoring group were the algorithms that did not really train and the random player. Both of these reached a median score of around 2300. This shows that even without really doing any thinking, a score above 2000 can be expected.

The second group is the group were type matching is done directly or indirectly. This includes Linear Regression, Linear Regression with more info, Monte Carlo Circle and Type Matching. All of these approaches had a median score between 3500 and 4000. This shows that simply dropping circles on circles of the same type is quite a good strategy, but it is not the optimal strategy.

The best performing group is the Monte Carlo group. This includes Monte Carlo Combined, Random and steps. All of these approaches simulate what would happen if a circle were to be dropped at multiple locations. The location that gave the highest score is then chosen and a circle is dropped at that location. All of these approaches reached a median score between 6000 and 7000. The strength of Monte Carlo lies in the simulation, which ensures that the ideal turn is not predicted, but instead selected based upon the outcome of the turn.

Lastly, the human player reached a median score a little over 5000. This shows that a human

is smarter than just dropping circles on circles of the same type, and does have the ability to strategically play the game. Where the human player falls short compared to Monte Carlo is that it does not have the knowledge to know what is really going to happen after dropping a circle. The human player will have to predict this, while the Monte Carlo player will know exactly what is going to happen. Therefore the human gets beaten by the Monte Carlo Players.

These results show that a score of a little above 2000 can be expected when not really using a strategy. When doing type matching, the score improves to 3500 to 4000, but ultimately the best strategy is simulation, which managed to reach a score of 6000 to 7000.

Where simulation excels, is that it knows precisely what is going to happen after the circle is dropped. This allows the player to play moves that a human player would never think of. When using simulation, it can be seen that the player uses the dropped circle to nudge and move other circles, which eventually end up colliding and matching. Simulation is also quite good at allowing multiple circles to combine in a single turn. This chain-reaction can be set up by putting circles of successive types against each other. If a chain reaction occurs in a turn that is simulated, it is very likely that that move will be played in the real game, since chain reactions usually increase the score by a lot.

It is interesting to see that machine learning models were not able to train themselves on this problem. Only the Linear Regression with pre-made data was able to accurately learn to drop circles on top of other circles.

For the Deep Q learning model that uses the topview to predict a move, it is possible that an input of 500 pixels and an output of 500 pixels is too much to quickly train with a model like this. For the Q learning approach, it is possible that the state does not describe the game well enough. Since a state can only be evaluated once a ball is dropped, it is hard to evaluate moving to the left and to the right separately from dropping the ball.

References

Browne, Cameron B. et al. (2012). "A Survey of Monte Carlo Tree Search Methods". In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1, pp. 1–43. DOI: 10.1109/TCIAIG.2012.2186810.

Janocha, Katarzyna and Wojciech Marian Czarnecki (2017). *On Loss Functions for Deep Neural Networks in Classification*. arXiv: 1702.05659 [cs.LG]. URL: https://arxiv.org/abs/1702.05659.

Kaelbling, L P, M L Littman, and A W Moore (May 1996). "Reinforcement learning: A survey". In: *J. Artif. Intell. Res.* 4, pp. 237–285.

Kroese, Dirk P. et al. (2014). "Why the Monte Carlo method is so important today". In: *WIREs Computational Statistics* 6.6, pp. 386–392. DOI: https://doi.org/10.1002/wics.1314. eprint: https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/wics.1314. URL: https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/wics.1314.

ktrZetto (2023). *The Most Over-Complicated Suika Game Techniques*. URL: https://www.youtube.com/watch?v=bS0C8vrBiGo.

Loshchilov, Ilya and Frank Hutter (2019). *Decoupled Weight Decay Regularization*. arXiv: 1711.05101 [cs.LG]. URL: https://arxiv.org/abs/1711.05101.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Alex Graves, et al. (2013). "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602. arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Andrei A Rusu, et al. (Feb. 2015). "Human-level control through deep reinforcement learning". en. In: *Nature* 518.7540, pp. 529–533.

Rodgers, Philip and John Levine (2014). "An investigation into 2048 AI strategies". In: *2014 IEEE Conference on Computational Intelligence and Games*, pp. 1–2. DOI: 10.1109/CIG.2014.6932920.

Shao, Kun et al. (2019). "A survey of deep reinforcement learning in video games". In: eprint: 1912.10944 (cs.MA).

Silver, David et al. (2018). "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* 362.6419, pp. 1140–1144. DOI: 10.1126/science.aar6404. eprint: https://www.science.org/doi/pdf/10.1126/science.aar6404. URL: https://www.science.org/doi/abs/10.1126/science.aar6404.

Şimşek, Simón Algorta; Özgür (2019). *The Game of Tetris in Machine Learning*. URL: https://arxiv.org/abs/1905.01652.

Takahara, Kanako (2023). *Suika Game: How a Japanese home projector app became a viral hit*. URL: https://www.japantimes.co.jp/business/2023/12/25/tech/suika-game-aladdin-x/.

Watkins, Christopher J C H and Peter Dayan (May 1992). "Q-learning". en. In: *Mach. Learn.* 8.3-4, pp. 279–292.