Machine Learning Engineer Nanodegree
Final Report

Sebastian Schmitz

May 30, 2018

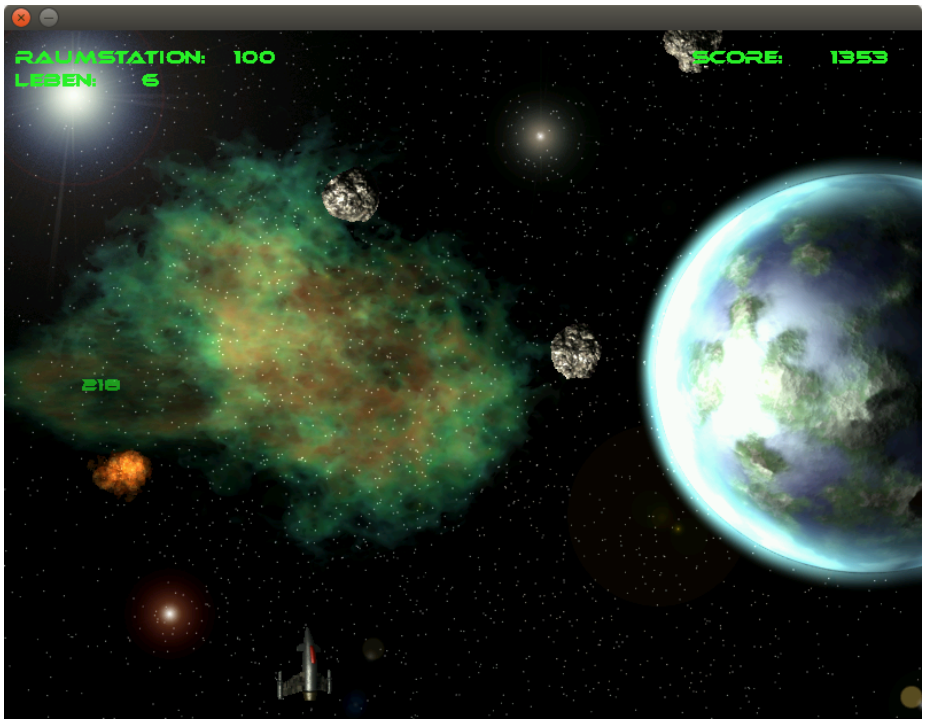

Figure 1: 2d space game

1 Definition

1.1 Project Overview

As an exercise to learn C++ and take a first peak into game programming, I programmed a 2d space game at the beginning of my studies in 2010, see figure 1.1. The goal of the game is to reach a score as high as possible by evading and shooting down asteroids. These asteroids spawn at random positions on the top of the screen and will fly towards the bottom of the screen in a vertical line. If they are not shot down they will either hit the player's ship making it lose a life or hit the off-screen space station which the player is supposed to protect. The space station gets repaired over time, but the game is over if its health drops below zero or the player loses all of his lives.

In this project I explored how well a machine learning system can grasp the problem of maximizing the score while playing this game. The goal was to give the applied algorithm the least amount of bias possible: it was supposed to find out on its own which actions are helpful and which are not.

My incentive to use unbiased approaches is the prospect of finding ways to play a game that are better than or just different from the approach a human would take to learn more games and other problems. In order to be able to showcase this, another goal of this project was to seamlessly integrate the solution into the existing game so that we can watch what the system learned and how well it performed.

1.2 Problem Statement

Before the start of this capstone project the game existed only as an interactive, GUI-based game playable using the keyboard. The first task was therefore not concerned with machine learning but with software engineering: The program had to be modularized to allow for a learning algorithm to use the game logic that is used when a human plays it. Special attention needed to be paid to the time that passes between two frames: As usual, a target number of frames per second was set. In every iteration of the main loop the game logic is applied: It moves the elements of the game according to the time that has passed since the last frame which is supposed to be 1/60th of a second. Afterwards, GUI elements are drawn on the screen and the thread then sleeps until it is time for the next iteration. The same needed to be true when training the algorithm without a GUI. Hence, a fake timer was implemented that always said the time between frames was 1/60th of a second. This way, the algorithm could be trained without a GUI in much shorter time and the results would be applicable later even in a GUI run.

The problem statement for the machine learning part of this project is a little more intricate. Nowadays, reinforcement learning algorithms combined with convolutional neural networks are powerful enough to learn games only from sensory input[2]. Thus, a decision had to be made regarding what information to give to the algorithm: It could either use the actual data the game logic works with or the visual representation of this data. However, since this project is not supposed to compete with a Google project - neither regarding manpower nor computational resources - the decision was made to use the data that is already there:

- Lives left: Every time the space ship gets hit by an asteroid, the player will lose a life, and eventually the entire game if no lives are left. The player starts with six lives.
- Space station health: Every time an asteroid is allowed to cross the screen, the space station will lose 16 health points and regenerate it at a fixed rate of five points per second until it is back at 100 again. If the space station health drops below 0, the player will lose the game.
- Asteroids: Set of x- and y-coordinates describing the positions of the asteroids.
- Ship position: The ship can move left and right on the lower edge of the screen.
- Weapons array cooldown: 0.5 seconds need to have passed between shots.
- Shot positions: Set of x- and y-coordinates of the shots the player has already fired.

After specifying the data the algorithm can work on we can now define the task at hand using the insights from [1]:

In each frame the algorithm is presented with the data explained above - the so-called *state* of the game - and has to make a decision what to do next

- taking a so-called *action*. To do so, we need to find a deterministic policy π that yields an action a for every given state s:

$$\pi(s) = a$$

However, we do not just want to find *any* such policy, we want to find one that maximizes the score before the game is over. Therefore, we define a value function V so that

$$V_\pi(s) = E[R] = E[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s]$$

. R is the expected return following policy π starting in state s. Two more things are needed to fully grasp this formula: The reward r_t and the discount factor γ. The former is the direct return for an action given by the environment and can be positive or negative. The latter is a value between 0 and 1 and determines how strongly future rewards are taken into account.

The task to solve in this setting is to find a policy π which maximizes $V^\pi(s)$:

$$V^*(s) = \max_\pi V^\pi(s)$$

Since we cannot know the number of policies it is computationally unfeasible to directly calculate the optimal policy.

Hence, an algorithm is needed that can approximate the optimal policy in an iterative fashion. It will start with an initial guess at the optimal policy and slowly improve its performance until it converges.

1.3 Metrics

In contrast to supervised learning problems where you can easily measure the performance using precision, recall, or F-scores, the performance of the proposed algorithm is only measurable using feature offered by the application domain. Since the task of the algorithm is to maximize the score, that is the measurement we are going to use.

The score starts at 0 and increases every time the player shoots down an asteroid. The increase depends on the asteroid's position: The further it is away from the player in y-direction the higher it is. Shooting down an asteroid therefore yields between 64[1] and 600 points.

The algorithm will first train by playing the game until it has explored the state space sufficiently and has exploited the gained knowledge to optimize the policy. To make the analysis of the performance statistically sounder, the algorithm will then be tested 100 times and we will take note of the minimum and maximum score as well as the mean and standard deviation.

2 Analysis

2.1 Data Exploration

As described in the problem statement, there is no dataset to analyze. Rather, the game consists of several elements relevant to the game logic that make up

[1] Because it is not possible for the ship and an asteroid to overlap and the ship is 64 pixels high.

the state space. In detail, these are:

- Player lives: As described above, this is an integer ranging from 6 to 0.
- Space station health: As described above, this is an integer ranging from 100 to 0.
- Asteroids: The asteroids spawn at eight equidistant points off the top edge of the screen. The spawnpoint is chosen at random. In every subsequent frame they travel with a speed of 200 pixels per second towards the lower edge of the screen. Thus, in each frame there is a set of (x,y) coordinates describing where the asteroids are. Each asteroid is 64 pixels wide and 64 pixels high.
- Ship position: The player can be at any point on the lower corner of the screen as long as it does not crash into the border of the screen.
- Weapons array cooldown: A floating point number keeping track of how long ago the last shot was fired. The player can only fire if the last shot was more than half a second ago.
- Shot position: When the player shoots, a shot spawns at the ship's position and travels up in a vertical line with a speed of 400 pixels per second until it hits an asteroid or eventually goes off-screen and gets deleted.

Furthermore, there are details to the game logic that are not directly observable and that don't change over time:

- The speeds the player and the asteroids travel at, which is 300 and 200 pixels per second, respectively.
- The amount of health the space station loses when getting hit by an asteroid, here: 16 health points per hit.
- The rate at which the space station gets repaired, here: One health point every 0.35 seconds.
- The rate at which the asteroids spawn, here: One every 0.75 seconds.
- The size of the screen is 800 by 600 pixels.

Since this second set of information is not directly conveyed to the human player either, it is not supposed to enter into the state space. We will therefore focus on how the first set of information forms the state space and make a guess at how large it is to estimate the difficulty of the task at hand.

The combination of the player's lives and space station health brings the state space to a size of $7 \cdot 101 = 707$.

The combinations of asteroids are a little harder to calculate: They spawn every 0.75 seconds and need $\frac{600}{200} = 3$ seconds to travel to the bottom of the screen. Since they spawn a little above the screen - at a y-position of -60 - and crash into the space station at y-position 590, there can be at maximum five asteroids at once. Since there is a 0.75 pause between asteroids, the cardinality is not just the amount of different positions an asteroid can be in to the power of five. If there are five asteroids, the one that spawned earliest is at least at

y-position 540: It spawned at -60, then three seconds passed making it travel to 540 and making four more asteroids spawned during that time. The first asteroid can be in $(590 - 540) \cdot 8 = 400$ positions. The y-positions of every subsequent asteroid is irrelevant because it is directly dependent on the first asteroids y-positions. Therefore, only their x-positions matter. This brings the state space spanned by the asteroids to $400 \cdot 8 \cdot 8 \cdot 8 \cdot 8 = 1,638,400$ different combinations.

The ships position ranges from 0 to 752[2].

The weapons array cooldown is a floating point number that can in theory take any positive value. However, if we assume that the player fires a shot at the latest one second after the last shot was fired - so half a second after it is already eligible to fire another shot - we can limit the combinations to 60, because there are 60 frames in a second.

A shot can - in principal - be in any position of the screen, as long as it does not overlap with an asteroid. Since a shot travels 400 pixels per second and two shots can be taken per second, there can be three shots at the same time. Each can roughly be at $200 \cdot 600 = 120000$ positions, bringing the combinations in total to a staggering number of 1728000000000000.

This leads to a total size of the state space of $7 \cdot 101 \cdot 1638400 \cdot 753 \cdot 60 \cdot (600 \cdot 200)^3 = 90,433,495,498,752,000,000,000,000,000L = 90E27$.

Obviously, this is much too big to be handled in a project of this scope, making it crucial to find a way to reduce the size of the state space without losing too much information or biasing the algorithm. How this is achieved will be explained in the chapter Data Preprocessing.

In addition to the state, the agent also has access to two further kinds of data: The reward it gets from the environment and the actions it can take in a given state.

The reward is not to be confused with the points the player gets for shooting down an asteroid: This increase in score can serve as a reward, but it is not necessarily the only feedback the player gets. Furthermore, it does not even have to included in the reward at all: the player's task is to maximize the score. Incentivizing it to hit targets seems to be a good way to achieve this goal. But it is also possible that the agent learns to maximize the score by learning to stay alive. The longer it stays alive, the higher the score is. Thus, the agent will get punished when losing a life, the space station getting hit and losing the game. These rewards and punishments served as a good starting points as explained in the chapter Refinement and were later expanded by another reward. The exact values can be found in the chapter Model Evaluation and Validation.

The possible actions in a state are straight forward to explain: The player can either shoot, move right or move left. If the player is not on the right border, it can move right. The same is true for moving left. The ship can only shoot after at least half a second has passed since the last shot in order to avoid permanent fire.

[2]This is a programming error. It should only be able to fly to an x-position of 736, because it is 64 pixels wide. However, in a previous version of the game, the sprite for the ship was 48 by 48 pixels and the maximum x-position was never changed after.

2.2 Exploratory Visualization

Since this project is about a game, every frame of the game is a visualization of the data we are to examine. Therefore, the example screenshot 1.1 will be analyzed. We can see that there are three asteroids which spawned in the middle region of the screen. The vertical distance between them is the same since they spawned at a fixed interval. One asteroid was shot down recently: The explosion caused by that is still visible and it yielded a score of 218. The player and the space station are both still at full health and the player has reached a score of 1353 points so far.

2.3 Algorithms and Techniques

In the MLND lectures we learned about three fields of machine learning: Supervised learning, unsupervised learning, and reinforcement learning. The proposed problem does not include large datasets that we can mine for information which we would need to apply (un)supervised learning methods. It is rather an environment in which an algorithm has to learn over time, adapt to so far unseen situations and generalize the experiences made in order to be able to handle as many problems that can arise within this environment as possible.

The solution will therefore leverage the strength of algorithms coming from the field of reinforcement learning: The system will start with no knowledge about the problem whatsoever, but it will perceive the state of the environment and will be able to derive a set of actions from this state. At first, the actions taken will be randomly chosen: The environment transitions into another state and the agent will immediately receive feedback on whether the choice of action was good or not. From these feedbacks the algorithm will over time create a graph modelling the environment: The nodes will be the states, the edges and their weights will be the actions. Since the goal is to learn how to behave, the level of randomness when choosing actions will be decreased by a small amount with every run. This leads to a transition in the behavior: In the beginning, the agent will explore the environment and after some time start exploiting its knowledge to ensure that the best solution was found.

The result will be an agent that has explored a big portion of the state space and can play the game reasonably well, ideally better than a human player.

Early experiments showed that it did not make a big difference whether an on-policy method like SARSA[3] or the off-policy method Q-learning[4] was used. This work will therefore focus on Q-learning, a method which aims to solve the problem described in the problem statement. In each step, the value, Q_t, of an action, a_t, the player can take in a state, s_t, is updated according to the following formula:

$$Q_{t+1}(s_t, a_t) \leftarrow Q_t(s_t, a_t) + \alpha \cdot (R_{t+1} + \gamma \cdot \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t))$$

Q is the value of an action taken in a state. This is what the player will try to maximize over time. α is the learning rate between 0 and 1. The higher it is, the more the current value will change, as one can easily see in the formula: if it is set to 0, the term in the parentheses does not have any weight at all

[3] `https://en.wikipedia.org/wiki/State-action-reward-state-action`

[4] `https://en.wikipedia.org/wiki/Q-learning`

and the value does not change. If set to 1 the current value will be completely forgotten. γ is the discount factor between 0 and 1. As explained in the Problem Statement, this factor determines how strongly the future rewards are taken into account. A high gamma makes the agent strive for more long-termin solutions. Q-Learning is called an off-policy method, because it always takes the maximum action value from the next state into account even though it might not take that action after moving to that state: With a certain probability ϵ it will take a random action.

In order for the agent to first explore the state space and then increasingly exploit its knowledge, we will implement two different functions for the ϵ-value which determines the likelihood of choosing a random action at every time step: One function, $l_\epsilon(t)$, will linearly decrease epsilon with every step, the second, $e_\epsilon(t)$, lowers it exponentially:

$$l_\epsilon(t) = 1 - \frac{t}{N}$$

$$e_\epsilon(t) = \epsilon_{95}^{\frac{t}{N*0.95}}$$

N here is the total number of training runs and e_{95} is the epsilon value after 95% of the runs are done, which was set to 0.000001.

2.4 Benchmark

Since the task of the agent is to maximize the score and there is no ground truth to what it can and should achieve, we can only compare it to the performance of other players: Human players will be asked to play the game to the best of their ability. We will take note of the statistical properties of the scores they achieve the same way we do for the trained player. However, in order to save time, human players will not be asked to play 100 times: 5 runs will have to suffice. Furthermore, a random agent will play the game in order to find a bottom line for the score count. The proposed system will be compared to these players and is expected to perform much better than the random player and hopefully as good as humans or even better.

3 Methodology

3.1 Data Preprocessing

As discussed in the chapter Data Exploration the state space of the game is too large to let a computer agent draw knowledge from it. The need for computational power and storage would both exceed the scope of this project by far. Furthermore, the true state space is probably a subspace of what was explained above with much fewer dimensions. *True state space* here means the space that needs to be explored in order to achieve a reasonable performance. The distinction between many states in the original state space does not yield any knowledge that can be exploited. Here are a few examples to highlight this:

- If a shot is going to travel off-screen without a chance of hitting another asteroid because it will leave the game before the asteroid can spawn, the position of that shot does not matter at all.

- If an asteroid is too far away from the ship in the x-axis and will hit the space station regardless of what the player does, the position of that asteroid also does not matter.

These were examples for information that makes no difference at all. There is more information that can probably be summarized: We will lose a bit of information, but the increase in performance due to the reduction of the problem's dimensionality might compensate for that:

- The exact number of the space station's health points will be quantized into four categories: 0 HP, 1 to 33 HP, 34 to 67 HP and more than 68 HP.
- The asteroids' positions will be boiled down to one piece of information: Where is the asteroid that will not get hit by a shot so far and that is still reachable for the player? This is stored in using two bits: It might be to the left, to the right, in front of the ship, or non-existent.
- Furthermore, the program will know if it will hit an asteroid if it shoots, moves left or moves right.
- The possible 60 states of the weapons array will be expressed in one bit: Is it possible to shoot or not?
- The player's position will take two bits: Can the ship move left, right, or both?
- The player's lives will be put into one bit answering the question: Does the player have more than one life left or not?

This leaves us with a theoretical state space size of $4 \cdot 4 \cdot 8 \cdot 2 \cdot 3 \cdot 2 = 1536$. However, some combinations are not possible: For instance, the next asteroid cannot be left of the ship if the ship is at the left border of the screen. Also, since the distance between asteroids is 100 pixels, it is not possible to be in a state where moving left or right both would mean crashing into an asteroid.

During evaluation the maximum number of explored states was 360, which is a reasonable number to handle and train with.

3.2 Implementation

As mentioned in the Problem Statement the first task was modularizing the already existing code base. Until this project began, the software engineering of the project was fashioned after the usual game development approach: There is a main loop that polls events, updates the game logic according to the events and the time passed since the last frame, redraws all the game components and sleeps until the next frame is supposed to begin to achieve a steady animation time regardless of computational power.

The following steps were taken to transform this game into a reinforcement learning testbed:

The player class was turned into an abstract class that arbitrary players can implement: A class implementing a human player would receive the events from the keyboard and hand them to the game logic. A reinforcement player would to one step of Q-learning and return the chosen action. Lastly, a random player would just return random actions regardless of the state it is in.

The renderer was also turned into an abstract class: Besides the actual SDL renderer, a fake renderer that does not render anything was written.

Furthermore, there is now also a fake timer. Using the fake timer and the fake renderer, the algorithm can be trained under the same circumstances it would be under when training using the GUI but much faster.

Two new frontends were added: One for batch training and one for batch testing. The training phase builds up a database with tables for environment descriptions, test case results, states and actions. The training phase then reads test cases, loads the appropriate state space and later write the score values back into the database.

The system was built on an Ubuntu 16.04 machine using the following libraries:

- SDL 1.2, SDL_tiff 2.0, SDL_image 1.2
- sqlite3
- boost_program_options

It can be compiled using `cmake` and `make`.

The coding process was rather straight-forward and no bigger complications were encountered. However, the game uses a very old code base from a time in which I only had a few months of coding experience. It might have been prudent to start from scratch with the goal of having several different player and renderer implementations in mind, because adding these functionalities to an old code base took a lot of time. The way it was carried out now does not make me 100% happy and the code needs some solid refactoring.

3.3 Refinement

One important aspect of a reinforcement learning system has gone almost unmentioned so far: The reward the agent receives from the environment after every action.

The initial idea was to give the RL agent only the reward a human player gets as well: The points from shooting down an asteroid. the maximum number of points for this is 600 and it goes down with every pixel the asteroid comes closer to the lower edge.

However, it was discovered that this is not enough to incentivize the agent to move enough and achieve higher scores: It would stand still and just keep shooting, eventually hitting some asteroids that randomly spawned in front of it. It was not aware that losing lives and the space station taking damage hurt its performance.

Hence, a big punishment was added for the terminal state and smaller ones for the space station getting hit as well as for losing lives. Still, the agent would not move. The agent would only start moving and playing the game in a proper way, after a punishment was implemented for the distance to the asteroid that is closest to the lower edge, still reachable and not already targeted by a shot.

In order to incentivize exploration, a very slight punishment was added if no other reward would be given for an action. This way, when confronted with a new state, the agent chooses one of the actions at random, since they are all the best. If nothing happens, that action will be worth less than the other actions and the next time the agent enters this state, it will not be chosen.

The final solution therefore is an agent that can sense the environment in up to 360 different ways and that receives positive and negative rewards for nothing happening, getting hit by an asteroid, shooting down an asteroid, losing all of its lives, the space station getting hit, the space station getting destroyed and not being right in front of the most dangerous asteroid.

In the following chapter we will discuss how the different ways of sensing the environment and different rewards influence the player's performance.

4 Results

4.1 Model Evaluation and Validation

The final model was chosen after the agent trained for a significant amount of time with a large variety of parameters: Rewards, parameters inherent to reinforcement learning, and ways to sense the environment were all tested in different combinations.

Rewards As mentioned before, the player gets a reward between 0 and 600 for shooting down an asteroid and a negative reward for every pixel it is away from the asteroid closest to the lower edge as described above. These are fixed amounts and all the other rewards are relative to them. The tested rewards were:

Getting hit by an asteroid:	0, -100 or -1000
Space station getting hit:	0, -80 or -160
No event:	0 or -1
Game over:	0, -500, -5000

Reinforcement Learning parameters As described in Algorithms and Techniques the gamma value would decrease according to two different functions. Furthermore, the gamma and the alpha value were set to 0.5 or 0.9 in each run.

State space quantization In order to get a basic understanding of its surroundings the agent will have sensed in every training run what actions it could take and where the most dangerous asteroid is. Three additional quantization methods each including the knowledge of the preceding function were formulated: The second method tells the agent whether it will crash into an asteroid depending on the actions it can take. The third method informs it about its health and the last also includes the knowledge about the space station's health.

The different ways for the agent to perceive that game state, the rewards and the RL parameters allow for 1728 combinations. Every combination was tested and the results written into an SQLite database (`asteroids.db3`) which was uploaded to this repository.

The best result was achieved by an agent that ignored the amount of lives it still had as well as the space station's health. α and γ were both 0.5, the reward for no event was -1, it got -5000 when losing, -160 when the space station was hit and -100 when getting hit itself.

The agent was trained for 30000, because for many combinations the score curve started to plateau after 22000 games.

After training and identification of the best function to describe the state space and the best set of rewards and parameters, the agent was deployed into a different environment: screen size and the amount of spawnpoints were changed as well as the speed at which the ship travels. The first two variations did not hurt the performance of the agent much: it was still able to perform at an acceptable level and choses actions reasonably. However, when the speed of the ship was changed, suddenly the scores were much lower. This result is to be expected: The first two changes were filtered out by the quantization function. In other words, the agent does not care whether there are eight, six, or ten spawnpoints, since it only gets the information where the closest asteroid is. The same is true for the variation of the screen size. In contrast, changing the ship's speed had a detrimental effect: The agent had learned which action would lead to which following state. With an increased ship speed, the actions would not lead to the states they used to lead to, rendering a lot of knowledge void.

4.2 Justification

As explained above, we had two benchmarks to compare the performance of the agent to: Human players and a random agent. Two human players - one versed in computer games and one not so much - played the game for five runs each. The first player achieved an average score of 89352 points with a standard deviation of 9236. The lowest score was 76123 and the highest 101473. The second player achieved an average score of 32540 points with a standard deviation of 16699. The lowest score was 6918 and the highest 49023.

The random agent played 100 games and achieved an average score of 872 with a standard deviation of 435. The lowest score was 0 and the highest 2863.

The best performing RL agent on the other hand exceeded expectations by far: During the 100 test runs it got an average score of 116042 points. However, the standard deviation being 71380, the minimum score only 7541 and the maximum score 420518 indicates that it could still do a lot better.

Remarkably, among the ten agents that performed best, the only constant was the method they chose for state space quantization: It was always the second one, disregarding HP and lives even though it the agent could have made use of that.

There is no agent in the dataset that trained without the punishment for being far away from the most dangerous asteroid. However, during development this was the change that lead to the most significant improvement in performance.

The entire data is included in the repository of this work. If you want to retrace the steps taken here, make sure the database `asteroids.db3` is in the subfolder `ReinforcementLearningTraining` of your build folder and the testing frontend will load the data as needed. You can browse the database using an SQLite viewer, choose a `test_cases_id` and start the testing algorithm with that ID as parameter. It will load the appropriate states and play the game 100 times. If you don't specify an ID, it will load all the test cases.

5 Conclusion

5.1 Free-Form Visualization

As mentioned before: This is an algorithm that was trained to play a game. Hence, it is best if the reader did the visualization on her own: Build the game and start the frontend `AsteroidsGUI` with the parameter `406` (after placing the database in the appropriate folder as described above). This will load the best performing agent which will thereafter play the game once. As you can see, the agent has learned that it is beneficial to hover towards the asteroid that is closest to the lower edge and still reachable. Furthermore, it will not crash into other asteroids if they are on the way to the most dangerous asteroid and it will also shoot down other targets in its path.

If you have any trouble getting this to work, please feel free to contact me.

5.2 Reflection

In this work we presented an algorithm that is able to learn how to play a game that was initially intended to be played by humans exclusively. The goal was to let a machine learning algorithm figure out how to achieve a score as high as possibe without giving it too much information, because every information a developer can come up with would introduce some bias. This worked ato some extent: The state space was shrunk in a reasonable way which even made the agent robust to changes in the environment. Hoewever, the best performance was only reached after a crucial piece of information was deduced from the game state: The position of the most dangerous asteroid, meaning the asteroid that is closest to the lower edge of the screen, still reachable by the ship and not yet targeted by a shot. Only after the agent was punished for being too far away from this asteroid did it reach human levels of performance.

For me, personally, that was an important realization about reinforcement learning: The agent will only do things if it has incentive to do them. The initial setup would only give the agent *very* sparse rewards. Sometimes, it took 60 frames until a shot reach an asteroid. It is very hard for a reinforcement learning algorithm to deduce which action actually lead to the reward so many frames later. The introduction of the punishment described above changed the rewards from being sparse to dense: Every frame the agent would get information on how well it is performing and since the position of the most dangerous asteroid relative to the ship was part of the state space, it was able to figure out that being in front of the most dangerous asteroid is very beneficial.

The final model exceeds my expectations. I am very happy with what the algorithm can do, because it plays better than I can. But of course, there is still some room for improvement.

5.3 Improvement

If you observe how even the best performing agent plays, you will see that it sometimes misses a shot or does not reach an asteroid it is pursuing.

Missing a shot happens, because the agent only knows about the most dangerous asteroid. It has no knowledge about any other target. The only thing

it learned is that shooting is beneficial regardless of state[5]. Performance could probably be improved by telling the agent whether there is an asteroid in front of it.

The second mistake happens, because the agent is not - yet? - able to plan ahead: It will pursue the most dangerous asteroid no matter what. If there is an asteroid in the way, it will wait until that has passed. This behavior makes it lose time and eventually unable to reach the target it was aiming at.

The agent could therefore be improved with more knowledge about the state space and probably longer training runs to make the result more stable.

References

[1] R. S. Sutton and A. G. Barto. Reinforcement learning: An introduction. *IEEE Transactions on Neural Networks*, 9(5):1054–1054, Sep 1998.

[2] D. Silver A. Graves I. Antonoglou D. Wierstra M. Riedmiller V. Mnih, K. Kavukcuoglu. Playing atari with deep reinforcement learning. 2013.

[5] A rather grim lesson to take away from this project, if you ask me...