Machine Learning Engineer Nanodegree
Captone Proposal

Sebastian Schmitz

March 8, 2018

1 Domain background

Ever since Tesauro's backgammon agent TD-Gammon [6] proved that computer programs can perform as good as humans in certain games, machine learning algorithms have been applied in numerous games. This lead to advances on both sides: The machine learning community could refine and improve their algorithms and the gaming communities learned new ways to play.

TD-Gammon used openings that had wrongfully been regarded as inferior by the leading players in the world. After a few tournaments where players tested the more conservative moves chosen by TD-Gammon, they became the near-universal choice, because they are - in fact - superior[4].

One of the pinnacles of this development was reached when the AlphaGo computer program defeated the European Go champion Fan Hui on a full 19x19 board[2]. Until then, Go[1] had been regarded as too complex to be mastered by an artificial intelligence.

Fueled by this development artificial intelligence programs trained through various forms of reinforcement learning[3] have learned to fly helicopters [5], play soccer [**?**] and optimize stock trade execution [**?**].

Inspired by the ability of letting an agent learn how to thrive in an environment without being told what to do, I developed a reinforcement learning agent which solved a version of the game Rush Hour[1]. The game consists of a 2d board with wooden pieces on it. The goal is to bring one of the pieces into a special position only by moving the blocks left, right, up or down. My approach was able to find a solution that was a lot shorter than the solution that came with the game.

I now want to build upon this and move from a discrete problem to a continuous one.

2 Problem statement

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 2. The goal of the game is to reach a score as high as possible by surviving and shooting down asteroids. These asteroids spawn at random positions on the top of the screen and can either hit the player's ship making him 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 player will lose if its health drops below 0. It might therefore be wise to crash the ship into an asteroid if it cannot be stopped otherwise and the space station is at low health. In other situations it could be beneficial to let the asteroids hit the space station, since its health bar is still full. Additionally, the amount of shots the player can fire at a time is limited, so it's not feasable to just fire at will.

The task of this project is therefore to let a machine learning algorithm figure out how to achieve the highest score. In order not to bias the learning agent, no human-devised strategies will be programmed into the approach. This will lead to an algorithm that potentially uses methods a human would not have thought of and that can cope with any of the random situations, the game throws at it.

[1] `https://en.wikipedia.org/wiki/Rush_Hour_(board_game)`

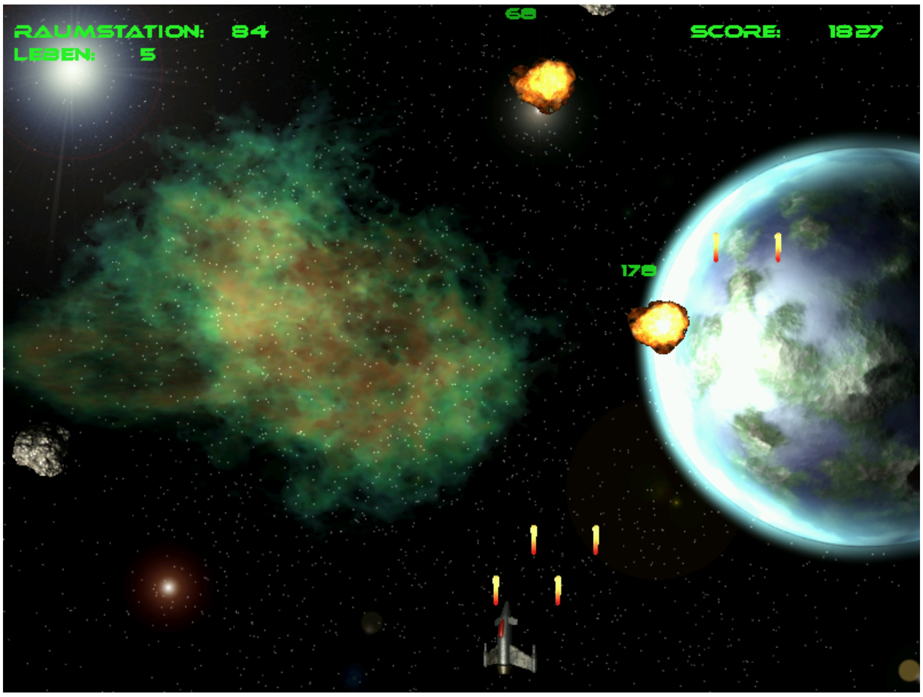

Figure 1: 2d space game

3 Dataset and Inputs

Since this is a simulation on a custom game there is no dataset to use. The inputs the algorithm can make use of are the following:

- Lifes 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.
- Space station health: Every time an asteroid is allowed to cross the screen, the space station will lose a set amount of health and regenerate it at a fixed rate until it's 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 corner of the screen
- Weapons array cooldown: A certain amount of time needs to have passed between shots.
- Shot positions: Set of x- and y-coordinates of the shots the player has already fired.

4 Solution statement

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.

5 Benchmark Model

Since the agent will train on a custom game, the choice of benchmark models and solutions to compare the performance to is sparse. Two options will be taken into consideration:

- A few human players will play the game several times each.
- A computer agent will execute random actions.

Ŵe will record the distribution of the scores from both groups.

6 Evalution Metrics

The automated agent will be implemented as described in the solution model and compared to the approaches from the benchmark model. The two options in the benchmark model will yield two sets of scores which will be used to evaluate the performance of the trained agent. It is expected that the human players will perform significantly better than the random agent. Therefore, the goal of the automated agent is to reach a score that is closer to the distribution of human players. If its performance remains at the level of the random agent, the approach will be considered a failure.

7 Project Design

So far, the project is a single player game without the option for a computer program to interact with it. The first thing to do is therefore to omplement an interface to the computer agent. The interface needs to provide the agent with everything it needs:

- A list of possible actions
- A desription of the state of the environment

After that interface has been impemented, the random action agent can be tested.

Like the reinforcement learning project this project needs an option to be executed without a graphical interface. If the state of the system needed to be rendered after every timestep, training an automated agent would not be feasable. To make sure that the results from the training are applicable to the real game, we will set a fixed framerate of 60 frames per second. After these additions, a reinforcement learning agent can start training.

The biggest challenge of this project will be finding ways of reducing the size of the state space without losing too much valuable information. Things that contribute to the state of the environment are:

- Position of the ship
- Position of asteroids
- Position of shots
- Health of the ship
- Health of the space station
- Cooldown of the weapons array

If the state space is too big, the agent will never be able to exploit knowledge from previous runs, because it will never reach states that have already been observed.

As in the reinforcement learning project from earlier we will work with different values for the exploration factor in the Q-learning setup. This time, we will also use different values for the discount factor γ, because striving for future rewards is exactly what this project is about: We want to find out whether crashing the ship into an asteroid and thus taking a short-term punishment can be worth it in the long run.

The final result will be a set of states and associated rewards that the agent has learned. We will be able to start the game with the graphical interface and to choose the agent to play the game which will then do so to the best of its abilities.

References

[1] Go (game). `https://en.wikipedia.org/wiki/Go_(game)`. Accessed: 2017-11-08.

[2] In a huge breakthrough, google's ai beats a top player at the game of go. `https://www.wired.com/2016/01/in-a-huge-breakthrough-googles-ai-beats-a-top-player-at-the-game-of-go/`. Accessed: 2017-11-08.

[3] Reinforcement learning. `https://en.wikipedia.org/wiki/Reinforcement_learning`. Accessed: 2017-11-11.

[4] Temporal difference learning and td-gammon. `http://www.bkgm.com/articles/tesauro/tdl.html`. Accessed: 2017-11-08.

[5] Pieter Abbeel, Adam Coates, Morgan Quigley, and Andrew Y. Ng. An application of reinforcement learning to aerobatic helicopter flight. In P. B. Schölkopf, J. C. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems 19*, pages 1–8. MIT Press, 2007.

[6] Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, March 1995.