Mastering AchtungDieKurve with Deep Q-Learning using OpenAI Gym

Erik Bohnsack and Adam Lilja

Abstract— **Achtung Die Kurve is a simple multi-player game played on a shared screen where the goal is to win over your competitors by surviving as long as possible. The goal of this project was to use reinforcement learning to train a bot to be able to play, and master, the game in both single and multiplayer mode. Some simplifications of the game were done to fit the scope of the project. The bot was trained with the Deep Q-Learning (DQN) algorithm from Open AI's Baselines using the OpenAi Gym. Only a multilayer perceptron (MLP) network in the DQN was evaluated. The bot was able to achieve superhuman performance skill-wise, but lacked any kind of tactical notion, making it an average opponent at best.**

I. INTRODUCTION

Achtung die Kurve, also known as Curve Fever, is a simple multi-player game, where each player is in control of a snake-like creature. The snake has a constant speed and is controlled by steering either left, right or do nothing to go straight forward. The goal is being the last survivor of each round. You die by running into either the surrounding walls or the trails of any player, including yourself. Playing the game one soon realize that it is easy to do simple mistakes, i.e. pressing the wrong button or mistaking your snake for an foe's. Especially when a stress- or meaningfull situation is before you. One solution to eliminate these risks of losing to your opponent could be making a bot that doesn't feel the pressure or making stupid mistakes. One way of approaching this is Reinforcement Learning (RL), as for instance has been seen in Deepmind's AlphaGo where a bot have mastered the two player strategy board game Go [11] [12]. Game playing with RL has recently gotten increasingly popular as computers' capabilities have sky rocketed and ToolKits for development and comparison of such algorithms have emerged, e.g. Vizdoom [13], RLPY[14], and OpenAI Gym[15]. RL refers to algorithms that unlike supervised learning don't need any specified input/output pairs to learn. Instead the algorithms learn themselves by exploring and exploiting and getting positive/negative feedback for different decisions while doing so. This project will aim at implementing a, by the authors created, game (very much like AchtungDieKurve) in the OpenAI gym environment, using RL to create an AI suitable for the game.

A. Related work

Using AI to achieve superhuman performance in games has been around for some time, e.g. the two famous cases of chess [16], which used extensive search methods, databases and at the time state-of-the-art hardware, and backgammon [18], which used a Neural Network (NN) evaluation function together with temporal difference learning. Narrowing down in the field of RL, there is a lot of available work implementing RL for similar games, using OpenAI Gym environment (Gym), e.g. [2]. [2] implements the game FlappyBird, and is able to reache superhuman performance with both DQN and DDQN, with DDQN giving better results. While snake exists in both Gym and as an PyGame Learning Environment (PLE), which can be used in Gym through the python library gym_ple, there were no implementation of AchtungDieKurve. However, as seen in the github-repo [3] it is possible to wrap a game as a Gym environment. It is possible to either use the output image as states in the Gym, e.g. used when training Atari games [4] or defining other, well chosen, states like [5] where an inverted pendulum is to be controlled.

B. Background theory

In RL there is typically not an input/output pair. Instead, as there is no one to tell you what is correct, the learner needs to discover itself what actions to take. This is done in with respect to maximize a numerical reward signal [1]. [1] formalizes the problem of RL as the optimal control of an incompletely-known Markov Decision Process (MDP), i.e. how an agent should behave optimally based on interaction with the environment. This is a very wide problem and can be solved in many ways, with various well-known algorithms. Dynamic programming (DP) is a set of iterative algorithms which finds the optimal policy given a perfect model of the environment. To be more specific, DP finds the optimal value function $v^*(s)$, the value of being in a certain state, and the optimal action-value function $q^*(a, s)$, the value of choosing an action a in state s, by iterative solving of the Bellman equations which in turn gives the optimal policy π^*[1]. However, problems rarely include perfectly known environments, instead the agent can make observations of parts of the environment. From this perspective, both Monte Carlo methods (MC) and Temporal Difference Learning (TD)[1] are algorithms that without knowledge about the environment, learns from experience of interaction between agent and the environment by estimating the action-value function $q(a, s)$. The difference between the two is that TD learns during an on-going sampling episode, so called online learning, while MC only learns after an episodes completion. There is also the concept of on-policy and off-policy learning, where on-policy is learning from the policy that is used to make decisions, while off-policy is learning from watching a different policy making decisions. TD-learning has advantages compared to MC, such as not having issues where problems have incomplete or very long episodes. Thus, there are many well-known algorithms for RL applications, e.g. SARSA, DQN, DDPG and A3C [17],

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
      $\Delta \leftarrow 0$
      For each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s))\big[r + \gamma V(s')\big]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      $a \leftarrow \pi(s)$
      $\pi(s) \leftarrow \text{argmax}_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
      If $a \neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return V and π; else go to 2

Fig. 1: Psuedo code Q-learning algorithm. Source [7]

which all stems from TD-learning. In this report our focus is on Deep Q-learning (DQN), which is a model-free, off-policy algorithm using a NN approximating the operator; Q-value. Note that both the action space and state space are discrete [6]. This algorithm is already implemented among Open AI's Baselines and the pseudo code can be seen in Figure 1

II. METHOD

A. Environment model

As already mentioned, the game did not exist in either OpenAI Gym or PLE. However, there existed a python-based AchtungDieKurve look-alike called FarBy [8]. This together with Gym wrappers such as [3], [9], helped when modelling the game from scratch as a PyGame wrapper of an OpenAI Gym environment. Using OpenAI gym enables not only the possibility of using their baseline networks[10] while training, but it also manages the underlying training infrastructure. To reach convergence in training within the scope of the project, two major simplifications were made. First off, the opponents were removed i.e the game of the goal was just to survive as long as possible. Secondly, instead of using frames of the games as game states/input to the network, the states were defined as in equation (1).

$$s = \begin{bmatrix} x & y & \theta & d_1 & \dots & d_n \end{bmatrix} \tag{1}$$

x is the position in horizontal plane, y vertical (origo in top left corner), θ the angle anti-clock-wise from the x-axis in degrees. d_i to d_n is distances to the closest objects in certain angles from the position x, y. $n = 9$ beams were used with equal spacing going from -120 degrees to 120 degrees in relation to the angle of the snake, as seen in Figure 2. The size of the board is set to 480 by 480 pixels and the speed of all snakes is 2 pixels/tick and the radius of all players is set to 2 pixels. The angle is in degrees and adjusted to always being the range $0-360$ degrees. The possible starting angles are discrete such that only fixed 10 degree steps are allowed. The beams have a predefined range, beam sight, in all runs below it is set to be 240 pixels, i.e. if no object is found closer than that we set the bounce distance being equal to the beam sight.

The action space, seen in equation (2), simply represents turning left, turning right and choosing to go straight ahead. Game-wise, taking action left or right represents a change in the angle of the head of the snake by 10 degrees respectively.

$$\mathcal{A} = \begin{bmatrix} \text{left} & \text{right} & \text{no operation} \end{bmatrix} \tag{2}$$

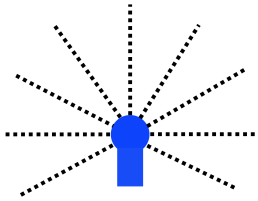


Fig. 2: Ego player with beams. Each beam measures the distance until it hits some non-background

The reward function is defined such that surviving one step/frame yields adding a positive reward of $+1$ to the cumulative sum, whereas colliding with the walls or tail yields ± 0 and game over, i.e. no further rewards are obtained. This reward represents the goal of surviving as long as possible, as opposed to trying to kill the opponent (making it crash). This will of course yield a more defensive strategy, but possibly easier obtainable. In order to further investigate how an opponent will affect the learned policy we decided to train the DQN Agent in 2 different environments, namely:

1) **Singleplayer environment (SPE):** The Agent is completely alone in the environment and can only lose if crashing with itself or the wall.
2) **Multiplayer environment (MPE):** The Agent faces three semi-immortal (dies when hitting wall) Random Agents. DQN Agent lose if crashing with itself, an opponent or the wall.

B. Random Agent

The simplest possible agent taking an action $a_i \in \mathcal{A}$ at the state s with probability:

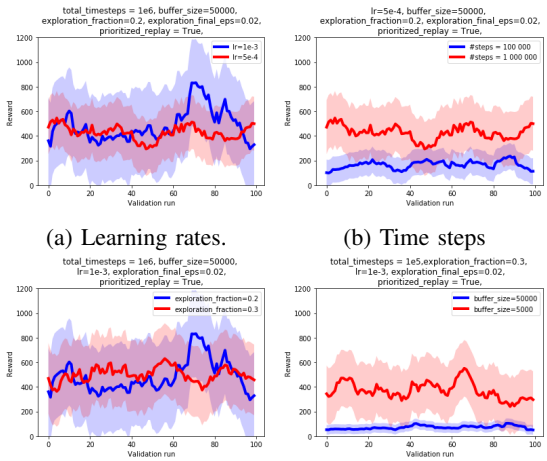
$$p(a_i|s) = \frac{1}{\text{\# actions}} = \frac{1}{3} \qquad \forall i = 0, 1, 2 \tag{3}$$

C. DQN Agent

In order to find an optimal DQN Agent different hyperparameters were tested. The outcomes of some of these tests are seen in Figure 3 and Figure 4. For the Singleplayer environment (SPE) the learning rate did not seem to affect the performance too much, neither did the exploration fraction. However, the number of training steps and the buffer size for the experience replay seem to affect the performance more. However, in the Multiplayer environment (MPE) the number of timesteps didn't change the performance remarkably. Probably due to the area in which the ego-snake can move becomes smaller due to the Random Agents (RA). Final hyperparameters can be seen in Table I. Note that the exploration fraction ϵ is the starting exploration fraction. This fraction is, as training progresses, decreased linearly until it hits a minimum value of $\epsilon = 0.02$. Furthermore, it's only the best performing policy that is being saved from the training, i.e. if the best policy is found after only half the training time that is still the one saved. Due to the risk of ending up in a "bad path", i.e. not leading to any good performance despite many training episodes, multiple (3-5) training runs were conducted with the same settings. Only the one with best performance of these were saved for further investigation.

Parameter	SPE	MPE
Total Timesteps	1e6	1e6
Learning Rate	5e-4	1e-3
Buffer Size	1e4	1e4
Exploration Fraction	0.2	0.2
Prioritized Replay	True	True

TABLE I: Hyperparameters for training the final versions of DQN SPE & MPE respectively.



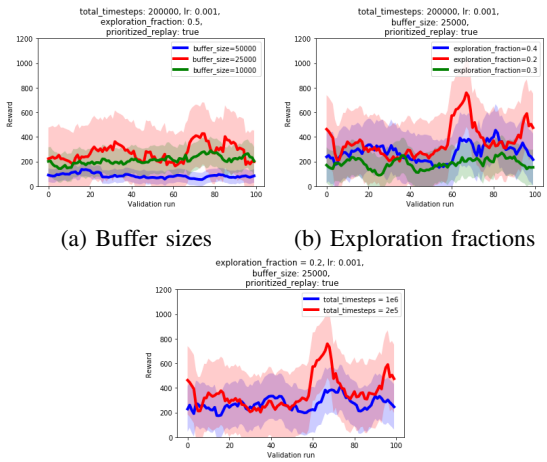
(a) Learning rates. (b) Time steps

(c) Exploration fractions (d) Buffer sizes

Fig. 3: Different tryouts of hyperparameters for SPE.

III. RESULTS & DISCUSSION

In order to understand how well our DQN Agent performs we compare it with the Random Agent (RA) and our (the authors) rewards over 100 games (blue, red and green respectively). The results for each run are displayed in Figure 5 for both SPE and MPE. In Table II the mean of the rewards can be found. It's seen that our DQN Agents are inferior to humans on average. However, on some occasions (especially for SPE) the DQN Agent outperforms a human player. The faded area is the standard deviation from the current value. It is seen that the standard deviation for the RA is fairly small, whereas for the DQN and human it is pretty high.



(a) Buffer sizes (b) Exploration fractions

(c) Time steps

Fig. 4: Different hyperparameters for MPE.

In Figure 6 we investigate how the DQN Agent thinks by looking at the q-values for each action as a function of the current frame. In this, pretty poor, run the player starts slightly above the center, headed north. It then makes a right u-turn and at frame A (after approximately 75 timesteps) it realizes it's about to run into the southern wall, thus the q-value for the 'forward' action drops and it takes a turn, in this case right. It overshoots the turn a bit and ends up in some pretty bad states and the q-values starts to fluctuate making the player's trajectory oscillate before proceeding straight north. When approaching frame C the agent was too slow at reacting and ends up in a very complicated position.

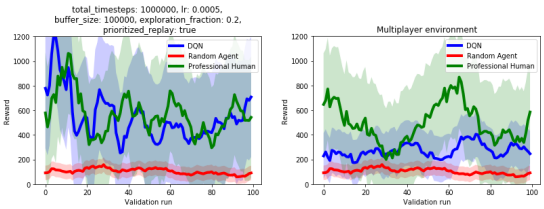
Some typical behaviour of the DQN Agent are seen in Figure 7. It seems like the agent wants to maximize the score through circling around, i.e. keeping close to one wall. Something that would make sense, as this is an efficient way of moving to take as little space as possible. However, eventually there is simply no space left and the game is lost. Except for making spontaneous mistakes in random states, the agent seems to do mistakes from time to time making it end up in a state from which it won't be able to recover. This is probably due to the lack of strategic thinking. The agent is not able to see what's behind a beam bounce and can thus not know whether there is much space left on the other side of the blockade or not. Using the entire board/image as input is of course a possible solution as it would be able to gain knowledge of the complete board.

As a further investigation each agent was dropped in the environment it was not trained for, SPE in MPE and vice versa. The results over 100 runs can be seen in Figure 8. The DQN Agent trained in SPE will on average get higher reward in both SPE and MPE. It will however also have a higher standard deviation. The reason for these phenomena is probably that the SPE DQN Agent is able to keep playing in the runs where the RA performs poorly, while on the occasions the RA happens to make a good run it will be worse.

Using the DQN network with MLP was satisfactory in regards to the scope of the project, as it clearly converged and gave a bot that showed promising characteristics. Unfortunately there was not time to continue to explore different networks and inputs. E.g. [2] mentions that DDQN gave better results and cut the training time. Continuing, using the full frame as input would have needed another network structure than MLP, i.e. add convolutional layers. One could also then imagine using more than one frame to gain the knowledge of speed/directions of itself and opponents. One could also extend to have the algorithm train against itself and modify the rewards to include positive feedback when winning over an opponent. One could speculate that this could add tactical characteristics to the bot.

Player	Singleplayer Mean Reward	Multiplayer Mean Reward
DQN Agent	544.33	271.61
Random Agent	104.43	100.75
Human	569.87	493.0

TABLE II: Single and Multiplayer environments. Mean rewards for 100 runs of DQN, Random Agent and a professional human player.

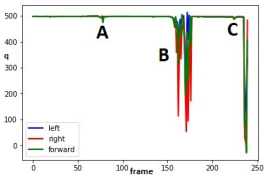


(a) Agent trained in SPE (b) Agent trained in MPE

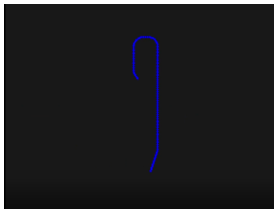
Fig. 5: Comparing the rewards of 100 games for DQN Agent, Random Agent and a human player.

IV. CONCLUSIONS

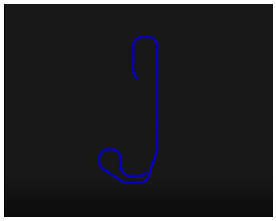
We developed a DQN Agent which on average is inferior to humans at playing AchtungDieKurve, but can on some occasions cut a dash and get an extremely high reward. The agent is very good at precise movement, making it able to run close to itself or the wall for a long time. It does, however, lack the strategic thinking. Something that could potentially be fixed by using the whole image as input to the network, but that is left as further work. This would require another NN than the MLP used in this report. It would be interesting to deploy and try other NNs and compare. Also would it be interesting to compare our agents with how state-of-the-art Path-planning, Genetic Algorithms, and MPC algorithms would've solved the problem.



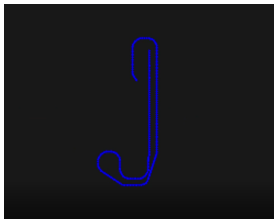
Q values for actions 'left', 'right' and 'forward' as a function of the current frame.



(a) At frame A the q value for 'forward' drops as the snake approaches the bottom wall.

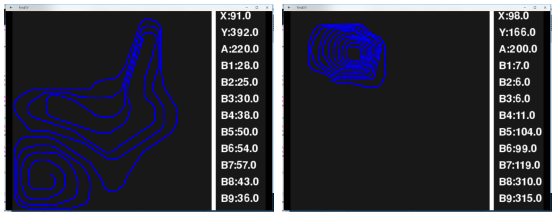


(b) At frame B the q value for 'right' and 'forward' drops as it will probably die taking one of those actions.

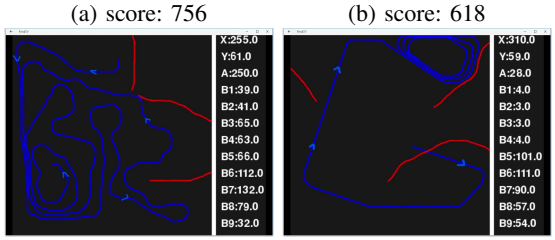


(c) At frame C the q value for all actions drop as we are in a bad position.

Fig. 6: Illustrative run for displaying q-values.

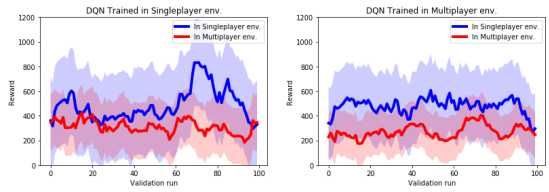


(a) score: 756 (b) score: 618



(c) score: 424 (d) score: 378

Fig. 7: Some runs that achieved higher score than average. The numbers displayed on the right side in each sub figure shows current position (x and y), angle (A), and the distances until bounce for each and every of the beams.



(a) SPE (b) MPE

Fig. 8: Comparing the rewards of 100 games for DQN Agents trained and tested in SPE & MPE respectively.

REFERENCES

[1] Richard S. Sutton, and Andrew G. Barto., Reinforcement learning: An introduction, 2nd edition, MIT press, 2018 (draft)

[2] Robert Chuchro and Deepak Gupta, "Game Playing with Deep Q-Learning using OpenAI Gym".

[3] Users: catherio and mhauskn, openai/gym-soccer https://github.com/openai/gym-soccer

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller, "Playing Atari with Deep Reinforcement Learning"

[5] AG Barto, RS Sutton and CW Anderson, "Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problem", IEEE Transactions on Systems, Man, and Cybernetics, 1983.

[6] Matiisen, Tambet (December 19, 2015). "Demystifying Deep Reinforcement Learning — Computational Neuroscience Lab". neuro.cs.ut.ee.

[7] "http://blog.csdn.net/songrotek/article/details/51378582"

[8] User: janowskipio, https://github.com/janowskipio/FarBy

[9] User: lusob, https://github.com/lusob/gym-ple

[10] https://github.com/openai/baselines

[11] https://ai.googleblog.com/2016/01/alphago-mastering-ancient-game-of-go.html

[12] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, Demis Hassabis. "Mastering the Game of Go without Human Knowledge"

[13] VIZDOOM : http://vizdoom.cs.put.edu.pl/

[14] RLPY : https://rlpy.readthedocs.io/en/latest/

[15] OpenAI Gym : https://gym.openai.com/

[16] Murray Campbell, Joseph Hoane Jr., Feng-hsiung Hsu (2002) "Deep Blue"

[17] Olivier Sigaud, "Reinforcement Learning From the basics to Deep RL", http://e.guigon.free.fr/data/masterMSR/deepRL.pdf

[18] Gerald Tesauro "Temporal Difference Learning and TD-Gammon" (1992)