Using Deep Reinforcement learning for creating an adapting agent playing zero-sum games

Eosandra Grund
egrund@uni-osnabrueck.de
Fabian Kirsch
fkirsch@uni-osnabrueck.de

Osnabrück University
April 3, 2023

Abstract

Deep reinforcement learning has proven to successfully train agents to play atari games [1] or even do tasks like robotic control [2]. This project explores the development of a deep reinforcement learning (DRL) agent, which is able to adapt to its opponent's performance and adjust the difficulty of itself accordingly. The result should be a balanced game. As a base for the adapting agent, a deep Q-network was trained to play Tic-Tac-Toe and Connect Four using self-play. The project then investigates the impact of different action decision formulas on the agent's performance. The results show that the win-to-lose-to-draw ratio can become way more balanced against opponents of different levels using these formulas instead of choosing the best action. But the balance becomes less with more complex games, therefore some improvements will be suggested. Overall, this project demonstrates the potential to use DRL agents as an approach for developing game-playing agents that can learn from scratch and adapt their performance to opponents, without relying on human-designed heuristics.

1 Introduction

In recent years, there has been a growing interest in the development of game-playing agents that can learn to play games from scratch, without any prior knowledge or heuristics. These agents use machine learning techniques such as deep reinforcement learning (DRL) to improve their game-playing abilities over time. Moreover, Mnih et al. [3] introduced a DRL learning model and a Q-learning variant that was able to outperform human experts in three out of seven Atari games with no adjustments done on architecture or hyperparameters. As a follow-up, Mnih et al. [1] have shown their developed deep Q-network (DQN) and compared its abilities against other methods [4][5]. It outperformed them in 43 of 49 cases. In addition, their method was performed on a professional level in 29 of them.

As an agent learning to play a two-player game always needs an opponent, self-play agents play against themselves to learn a game without prior knowledge of an opponent. This principle is used by Silver et al.(2017) when they introduced their 'AlphaZero' algorithm [6] which is a more general version of their previous 'AlphaGo' [7] and is able to achieve a superhuman level of play within 24 hours in games of Chess, Shogi, and Go. Starting from random play and no

prior knowledge except the game rules to defeating world-champion programs in all of them. To conclude, self-play agents are able to reach good performance in multiple games if set up right.

When creating games, one high priority is to make them satisfying for the game player. This is highly influenced by multiple variables and one of them is the game difficulty. On one hand, if the game is easy, the game becomes boring. On the other hand, if it is too difficult, players do not get the sense of achievement and success they are looking for and could become rather frustrated [8]. First and foremost, the traditional approach would be providing difficulty options the player can choose from. This fails with a great diversity of skill levels [9]. Dynamic game balancing is trying to solve this. It includes a wide range of approaches but each has to satisfy three requirements according to Andrade et al. [9]. To satisfy successful game balancing, the agent first has to identify and adapt to the player's initial level quickly. Secondly, it must quickly and closely track the player's skill level and adapt accordingly. Lastly, while adapting, the player is not meant to recognize this behavior, as a result, the game stays believable. The player´s skill level is thereby often referred to as the 'challenge function' [10].

Nebel et al.(2020) [11] explored the potential of adaptive elements in the learning process and found that participants reportedly showed a lowered feeling of shame, increased empathy, and behavioral engagement against these adapting agents and revealed a positive impact on learning performance and efficiency. In studies using reinforcement learning this problem was tackled by having a simple action decision function that takes the current life score of the agent and the opponent into account as a challenge function while playing a fighting game [12][10]. This approach showed to be effective. Their usability test additionally showed an influence on user satisfaction depending on the performance of the agent [9]. Moreover, the lack of predictability in some agents improved the experience overall additionally.

In this project, we will try to answer the question if it is possible to have an DRL agent that creates a balanced game against opponents with different skill levels, by only slightly changing the action decision formula, when no information about the state of the game is provided during the game. With this in mind, we will develop an adapting self-play agent for playing the games of Tic-Tac-Toe and Connect Four. These games were chosen for their simplicity which is beneficial considering our limited resources and time for the project. Aside from that, simple games should be an adequate starting point for investigating the potential of our approach for future research. Due to the good performances of DQNs, we decided to use a DQN as our agent. It should be strong enough to learn zero-sum games and is additionally simple to implement.

First, we will train a high-performance DQN agent that can learn to play these games from scratch, without any prior knowledge or heuristics. And then, we will use this high-performance DQN agent as a base for our adapting agents. The testing will be done against a high-performing agent with different epsilon values to simulate differently strong opponents. We hope to demonstrate the power and flexibility of simple adapting agents in game-playing regarding game balance. They could be able to adjust the game balance to every player individually which might impact their experiences positively.

All the underlying code of the project and the mentioned trained models will be available at our github.[1]

[1] https://github.com/egrund/Self-Play_DQN_Project.

2 Training a high-performance DQN Agent

We trained several agents but decided on one best-performing for Tic-Tac-Toe and one for Connect Four. In this section, only the hyperparameters and architectures of the best-performing ones for each game will be mentioned.

2.1 Methods

To train the normal deep-Q learning agent [3], a simple convolutional neural network (CNN) was implemented in Tensorflow [13] and TensorBoard [14] was used for the visualizations. The model's architecture for Tic-Tac-Toe contained one 2D convolutional layer with 128 3x3 filters, a global average pooling layer and then one hidden dense layer with 64 units and an output layer with 9 units using a linear activation function. For Connect Four we used the same architecture but 2 convolutional layers with each 128 4x4 filters and two hidden dense layers with each 64 units and a *tanh* activation function for the output layer with 7 units. Both models also used an Adam optimizer [15] with a learning rate of 0.001 and mean-squared-error loss. After each convolutional layer BatchNormalization [16] was used.

Additionally to the vanilla DQN, Polyak averaging [17] and prioritized experience replay [18] were implemented. Polyak averaging was done once for every iteration with an α of 0.9. The buffer contained a maximum of 100,000 samples. In the beginning, it was filled with at least 5000 samples. During each training iteration the buffer was filled with samples from 1024 games, and the agent was trained on 25,600 samples. The sampling games to fill the buffer at the beginning of the game were played against a random agent, which randomly samples one of the available actions. During the training, the agent played against an old version of itself having half the agent´s discount factor ϵ. The old agent was always the agent from the last iteration so before the last training step. ϵ was set to 1 initially and in each iteration 0.01% for Tic-Tac-Toe and 0.005% for Connect Four were removed. When it reached a value of 0.01 it stayed constant. The targets were calculated using a discount factor γ of 0.3, as the next reachable reward is always close in these games.

The Connect Four environment was taken from keras-gym [19] and uses the OpenAi Gym framework [20]. The observations have the shape of (7,7,2) and contain information about the last move and the current state of the grid. The rewards are 1 for winning, 0 for a draw and every move before the game is done, and -1 for losing. The Connect Four environment was modified to be used as a Tic-Tac-Toe environment. The rewards stayed the same but the observation with the shape of (3,3,2) does not contain information about the last move anymore.

2.2 Implementation

A UML class diagram with all classes directly involved in the training can be found in Figure A2. The training algorithm and the testing algorithms are implemented as functions. The training algorithm also works for training several agents at once that sample by playing against each other and themselves. Every few iterations the agent plays 100 games against a random agent to test its performance.

The sampling algorithm always samples from several games at once. It works with normal agents, adapting agents, one opponent for all games, or several different opponents. Because the Connect Four environment and therefore also the Tic-Tac-Toe environment always just return the reward for the last move made, they never return that you lost and they give the next state, which is from the opponent's view. A wrapper makes the environments suitable for self-play by returning the reward after the player's and the opponent's move to the player, also giving

information about losing the game. The environments can also be used to give a penalty for every move or even returning a penalty when given an unavailable action instead of throwing an error.

Additionally we made files to be able to play against a person or an agent.

2.3 Results

As we trained on rather simple problems, changes in most of the hyperparameters did not change the results, only the training process, but the final performance was similar and also achieved in similar timespans. The decisions for the final models were made by comparing models from different runs and iterations. The results of that can be found in Table A1 for Tic-Tac-Toe and in Table A2 for Connect Four.

The Tic-Tac-Toe agent was trained for 7859 iterations. Its best performance was achieved at iteration 5300.[2] Testing over 10,000 games resulted in a winning proportion of 94.53%, a losing proportion of 0.0 % and a draw proportion of 5.47% against a random agent. One can also see in Figure 1, that the agent started to perform very well already at iteration 1200. This is a very good result, as a draw is relatively easy to obtain in Tic-Tac-Toe, because e.g. when the agent starts and has two marks in one line as soon as possible, the probability for the random agent to prevent losing in this first case is already $\frac{1}{6}$.

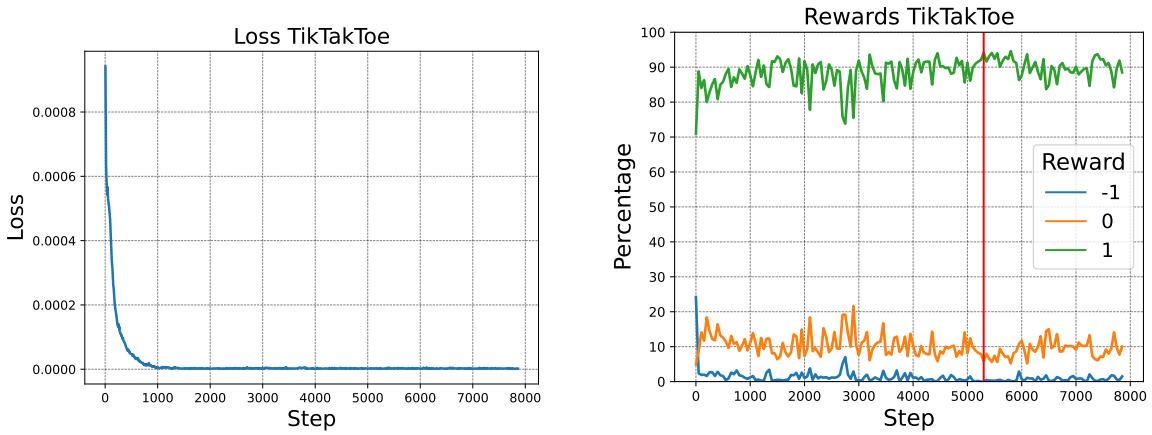

Figure 1: Left: Training loss of the high-performance DQN playing Tic-Tac-Toe.
Right: Percentage of different rewards while testing the high-performance DQN agent against a random agent playing Tic-Tac-Toe during training. The red line shows the chosen agent.
(-1 = losing, 0 = draw, 1 = winning)

The Connect Four Agent was trained for 900 iterations. Its best performance was achieved at iteration 400.[3] The training progress is plotted in Figure 2. Testing its performance over 10,000 games against a random agent resulted in a winning proportion of 99.52%, a losing proportion of 0.47%, and a draw proportion of 0.01%. This is a very good result, as in Connect Four, the beginning player with a perfect strategy will always win, therefore even a random agent has a small probability of winning.

[2] https://github.com/egrund/Self-Play_DQN_Project/tree/main/model/agent_linear_decay099
[3] https://github.com/egrund/Self-Play_DQN_Project/tree/main/model/agent_ConnectFour_tanh

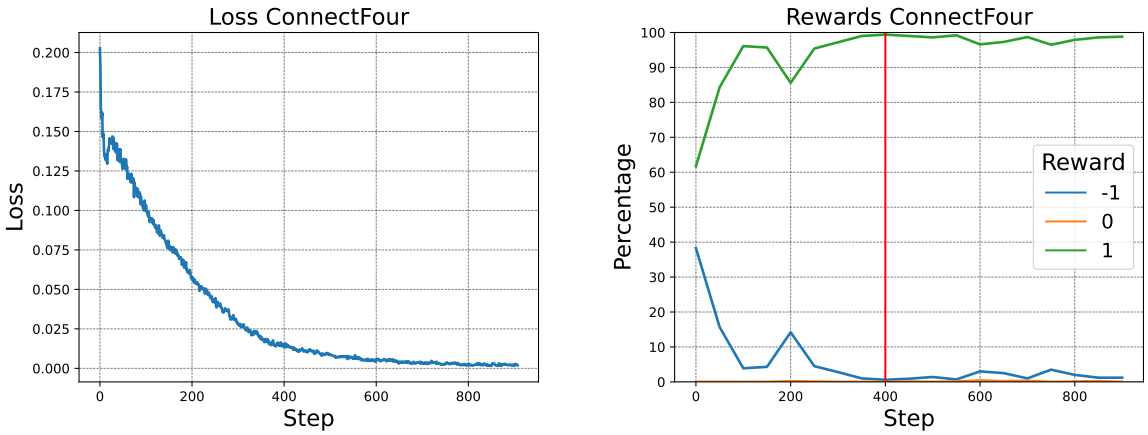

Figure 2: Left: Training loss of the high-performance DQN playing Connect Four.
Right: Percentage of different rewards while testing the high-performance DQN agent against a random agent playing Connect Four during training. The red line shows the chosen agent. (-1 = losing, 0 = draw, 1 = winning)

3 Adapting Agents

3.1 Methods

Simple Adapting Agents are obtained by using the high-performing DQN agent described in section 2, and changing its action decision formula. As the agent learned to predict Q_{sa} which is the expected future reward for a state-action pair, we can use this universal value differently than just choosing the highest value out of $Q_{s\vec{a}}$, as we do for the high-performing agent. The goal is to achieve a dynamically balanced game, with as few hyperparameters as possible to create a generally applicable method.

The simplest approach is to choose the action using Equation 1. This would be always the action with the expected future reward closest to 0.

$$\text{best}Q_{sa} = \min \sqrt{Q_{s\vec{a}}^2} \tag{1}$$

Where $Q_{s\vec{a}}$ is the vector of state-action values for each possible action a in state s. This is the output of the network. bestQ_{sa} is the chosen action. Because this is not completely balanced one can adjust it a little by scaling the positive values or negative values of $Q_{s\vec{a}}$, so that they are closer to zero. Which one to choose depends on the preference to win or lose of the agent using Equation 1.

$$\text{best}Q_{sa} = \min \begin{cases} Q_{sa} \times -1 & \text{for all } Q_{sa} < 0, Q_{sa} \in Q_{s\vec{a}} \\ Q_{sa} \times c & \text{for all } Q_{sa} > 0, Q_{sa} \in Q_{s\vec{a}} \end{cases} \tag{2}$$

$$\text{best}Q_{sa} = \min \begin{cases} Q_{sa} \times -c & \text{for all } Q_{sa} < 0, Q_{sa} \in Q_{s\vec{a}} \\ Q_{sa} & \text{for all } Q_{sa} > 0, Q_{sa} \in Q_{s\vec{a}} \end{cases} \tag{3}$$

Where c is a positive scaling value, $c \in]0, 1]$. Equation 2 shows the scaling of the positive and Equation 3 the scaling of the negative values of $Q_{s\vec{a}}$. This approach makes the agent always choose the closest scaled value to 0, if c is close enough to zero. Therefore you can only slightly direct the agents decision.

One further improvement might be to not just to take the action closest to zero but scale the actions around -1 to 1 and then use an action closest to a value $B \in [-1, 1]$.

$$\text{best}Q_{sa} = \min \sqrt{\left(\frac{Q_{s\vec{a}}}{\max \sqrt{Q_{s\vec{a}}^2}} - B \right)^2} \tag{4}$$

As B can be positive or negative, we can adapt the agent in both ways without much effort. But the adapting effect would be persistent. It does not really take the players level into account and therefore is not really adapting dynamically during playing. To achieve that, you have to make B an adaptable variable. For that, the average achieved reward $B_r \in [-1, 1]$ over the last m games was used instead of B, and can be seen as the challenge function.

$$B_r = \frac{\sum \vec{r}_i}{m}, \qquad i \in [-m, \ldots, -1] \tag{5}$$

$$\text{best}Q_{sa} = \min \sqrt{\left(\frac{Q_{s\vec{a}}}{\max \sqrt{Q_{s\vec{a}}^2}} - B_r \right)^2} \tag{6}$$

One last further improvement could be to scale B_r. For that, scaling factor $cp \in [0, \infty]$ is introduced. This change could increase the effect of Equation 6 if $cp \in]1, \infty]$ or decrease them if $cp \in]0, 1[$.

$$\text{best}Q_{sa} = \min \sqrt{\left(\frac{Q_{s\vec{a}}}{\max \sqrt{Q_{s\vec{a}}^2}} - cp \times B_r \right)^2} \tag{7}$$

If cp is very high, the agent would start to use its best or its worst action depending on the sign of Br. And therefore prefer to either lose quickly or win instead of playing draw. Scaling with cp could also be done with B in Equation 4. But because B and cp would be fixed, you could just enter a different B in the first place.

3.2 Implementation

A UML class diagram of the different subclasses of *Agent* can be found in Figure A3. The *AdaptingAgent* class and all its subclasses have the best agent, the game balance and the opponent level as attributes. Not all of the two latter are always used. A calculation value is introduced for all parameters that have to be set. It is not used, if it is not needed. To change the action decision function, the method *action_choice* is overwritten, and one other method is if it is needed.

3.3 Results

For Tic-Tac-Toe Equation 1 seems to create a balanced game already, as you can see in Figure 3. The percentages of winning and losing are almost the same. The results of Figure 3 and the results of the other equations for Tic-Tac-Toe can be found in Figure A1. They are all very similar, as the results for Equation 1 are already so good, which is the base of all the others too. You can also see in Figure A1, that the expected shifts in the proportion of winning, losing, and draws for different parameters and the different equations happened. Using Equation 2 created a stronger agent more similar to the results of the best agents while using Equation 3 let the agent prefer losing. Equation 4 did the same thing by increasing the losing proportion with $B = -0.1$ and

increasing the winning proportion with $B = 0.1$. Equation 6 instead does not need a value and is supposed to adapt by itself. It does look pretty similar to Equation 1 as do the results of using Equation 7. Even though the agent seems to prefer winning and losing a little more over a draw, which was expected too.

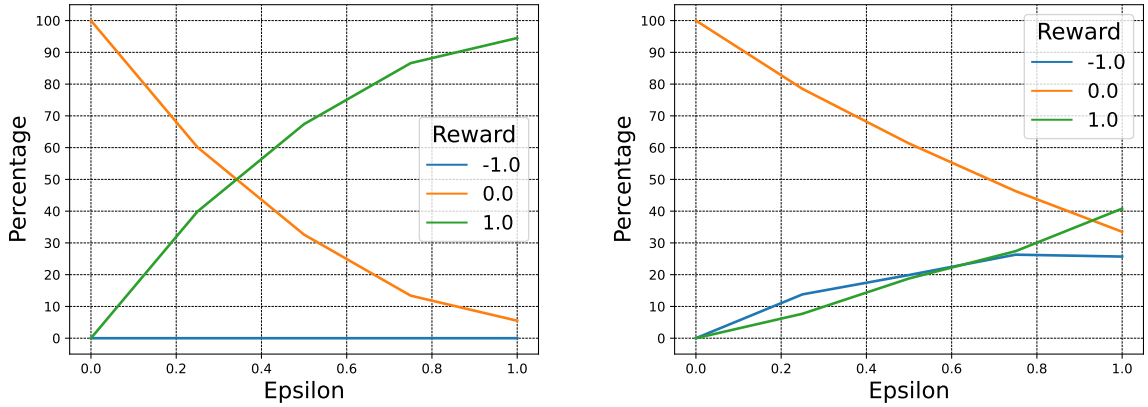

Figure 3: Showing the performance of different agents playing Tic-Tac-Toe against the best agent from section 2 using different ϵ values for the ϵ-greedy policy.
Left: The high-performance DQN Right: An adapting agent using equations 1.
(-1 = losing, 0 = draw, 1 = winning)

But for games with higher complexity Equation 1 might not work so easily, as one can already see through the results of Connect Four in Figure 4. There the values are not balanced at all. Therefore only changes with parameters that increase the winning percentage are depicted. As expected all of the changes improved the game balance, but only the results of Equation 7 can be considered balanced. The rest is not strong enough. An interesting behavior is seen using Equation 3 with $B = 0.5$. It looks balanced in total, but not at all for the individual levels, as the agent always loses against a strong opponent but wins over 70% of the time against a random opponent. This happens, because B does not change. It would have to be 1 to play balanced against a strong opponent and -1 to play balanced against a weak opponent. And therefore it plays good enough to play well against bad opponents but way too bad against strong opponents. When adapting B using Br in Equation 6, the win-to-lose-to-draw ratio is similar for all opponent levels, but the adaptation is not strong enough, as the agent still loses more than it wins.

Discussion and Conclusion

In this project, we first trained a DQN to play Tic-Tac-Toe and Connect Four which worked very well, as they achieved high performance. Then we used these agents as a base for adapting agents, which are supposed to keep the game balanced while playing. They had different action decision formulas, than the DQN agent originally, but used its Q-values. For Tic-Tac-Toe the different formulas achieved good results while only Equation 7 could achieve something balanced for Connect Four. This is because Connect Four is a way more complex game where it is easier to lose without the perfect strategy.

The final result is mostly balanced against more random opponents and needs improvements, especially against strong opponents. Presumed the opponent is best performing, and the high-performing underlying agent can compete. The agent adapts and plays a balanced game, then

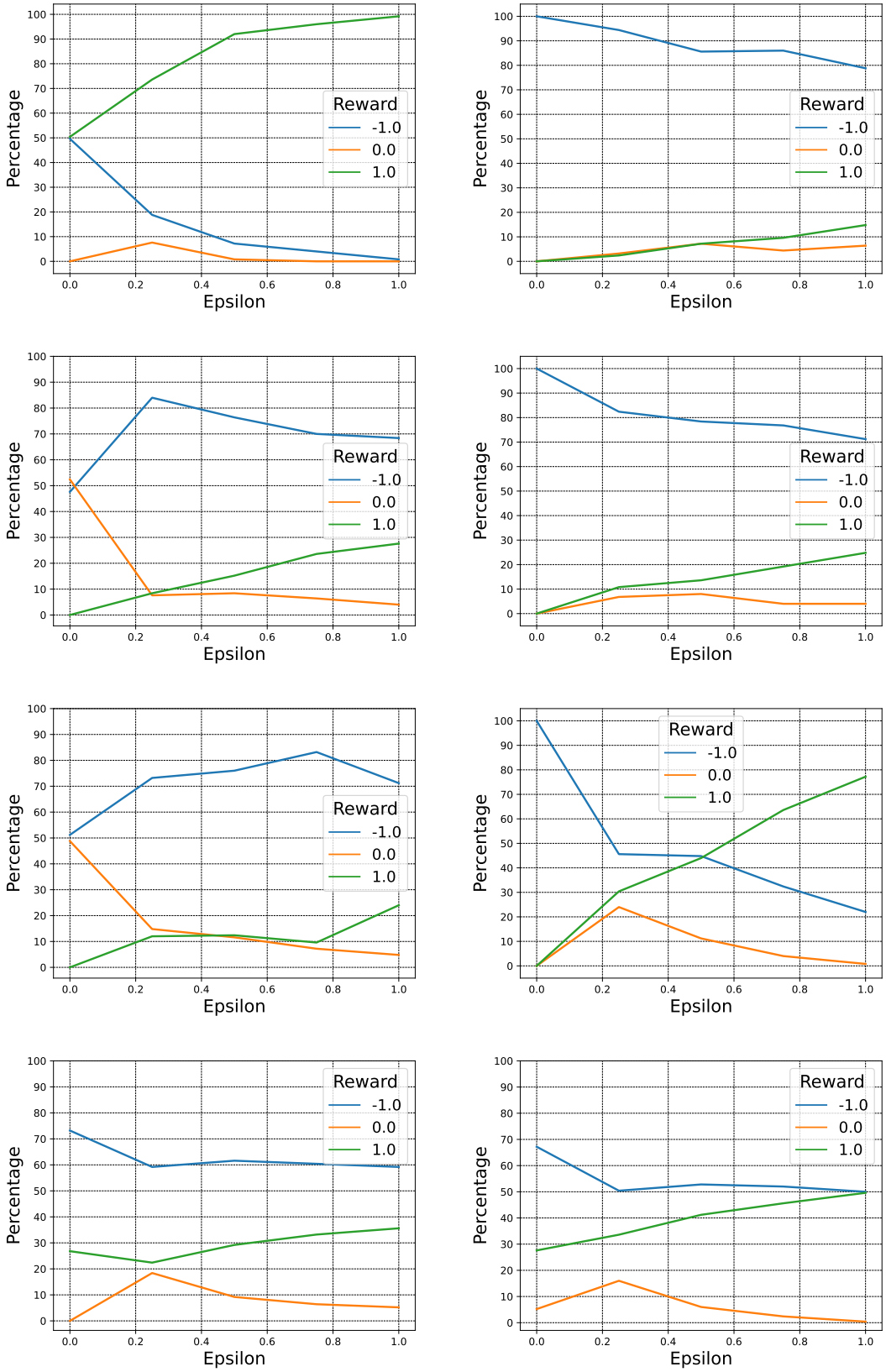

Figure 4: Showing the performance of different agents playing Connect Four against the best agent from section 4 using different ϵ values for the ϵ-greedy policy. (-1 = losing, 0 = draw, 1 = winning).
Top Left: High-performance DQN. Top Right: Adapting agent using Equation 1. Second Line: Adapting agent using Equation 2 with $c = 0.01$ on the Left and $c = 0.1$ on the Right. Third Line: Adapting agents using Equation 4 with $B = 0.05$ on the Left and $B = 0.5$ on the Right. Bottom Left: Adapting agent using Equation 6. Bottom Right: Adapting agent using Equation 7 with $cp = 3$.

the agent will pick actions closer to 0 again and lose until the balance is negative so it starts to pick better actions. The total game balance over a longer time therefore will not be zero but negative, because the adapting agent will be worse than the opponent. To prevent this, the agent would have to remember how it had to adapt at the beginning of the game to reach the balanced game, so the adapting does not only rely on the last few games. This could for example be achieved by remembering the game balance of the first few games until it crossed zero, instead of renewing it constantly.

Also, our agents do not really fulfill the first requirement by Andrade et al. [9], which is, to adapt quickly to the opponents level. They only use the average reward as information, which might work when playing lot's of short games, but will fail when playing few long games. Also the agent can adapt earliest in the second game played. This could be solved by using in-game information, but in games such as Tic-Tac-Toe specifically, there is barely information to do this. An agent directly trained to adapt could learn to use in-game information by itself.

To further test our method or improved versions one could test the methods for more complex games, as we used zero-sum games only. Instead of only checking the total win-to-lose-to-draw ratio, it would also be interesting to see its change throughout several games, when the agent adapts to the opponent. And if the ratio then is stable or fluctuating. This only accounts for formulas that depend on the average reward Br (Equation 6 and Equation 7).

An important follow-up question would be how natural the different action decision formulas feel for human players with different levels of expertise when playing against the adapting agents. This would correspond to the third condition for a good dynamically adapting agent according to Andrade et al. [9]. One could also start by doing a study on how the ratio has to be for different players so it feels balanced for them, and then adjust the method according to that. These values could change depending on the game, as different ratios are achievable in different games. One could also do a usability study similar to Nebel et al. [21] to test whether playing against a simple adapting agent improves the playing experience.

Alternatively to training a normal DQN and just changing the action decision formula, one could also train an agent directly to create dynamically balanced games. This could either be achieved by just using the different action decision formulas during training to create other samples or by changing the rewards. When changing the rewards one could maybe make the agent prefer draws instead of just a balanced game, which can solve the problem with Equation 6. There the agent prefers to win or lose instead of playing a draw in comparison to Equation 1. A little bit more complex approach would be to train the agent directly to try to predict the future game balance. When playing short games, it might be helpful to have a memory, as it might be difficult to read the opponent's level just from a few moves e.g. in Tic-Tac-Toe. There you should not give information about the current game balance in a way that the agent could just learn to calculate the target directly using the Q-values if you use the current game balance to calculate targets. One further idea could be to model the opponent directly in the agent's architecture similar to He et al. [22] did. This could make it easier for the agent to approximate the opponent's level by analyzing its moves directly.

References

1. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

2. Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373, 2016.

3. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

4. Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *CoRR*, abs/1207.4708, 2012.

5. Marc Bellemare, Joel Veness, and Michael Bowling. Investigating contingency awareness using atari 2600 games. *Proceedings of the AAAI Conference on Artificial Intelligence*, 26(1):864–871, Sep. 2021.

6. David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.

7. David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, and et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

8. Noah Falstein. The flow channel. *Game Developer Magazine*, 11(05), 2004.

9. Gustavo Andrade, Geber Ramalho, Alex Gomes, and Vincent Corruble. Dynamic game balancing: An evaluation of user satisfaction. pages 3–8, 01 2006.

10. Pedro Demasi and J de O Adriano. On-line coevolution for action games. *International Journal of Intelligent Games & Simulation*, 2(2), 2003.

11. Steve Nebel, Maik Beege, Sascha Schneider, and Günter Daniel Rey. Competitive agents and adaptive difficulty within educational video games. *Frontiers in Education*, 5, 2020.

12. Gustavo Andrade, Geber Ramalho, Hugo Santana, and Vincent Corruble. Extending reinforcement learning to provide dynamic game balancing. In *Proceedings of the Workshop on Reasoning, Representation, and Learning in Computer Games, 19th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 7–12, 2005.

13. Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

14. Mani Varadarajan, Billy Lamberta, William Chargin, Ryan James Walden, Asei Sugiyama, Mark Daoust, Manish Aradwad, Stanley Bileschi, and zac hopkinson. https://www.tensorflow.org/tensorboard/get_started, 2022.

15. Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

16. Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. pmlr, 2015.

17. Boris T Polyak and Anatoli B Juditsky. Acceleration of stochastic approximation by averaging. *SIAM journal on control and optimization*, 30(4):838–855, 1992.

18. Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

19. Kristian Holsheimer. keras-gym: Plug-n-play reinforcement learning in python with openai gym and keras., 2019.

20. Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

21. Steve Nebel, Maik Beege, Sascha Schneider, and Günter Daniel Rey. Competitive agents and adaptive difficulty within educational video games. In *Frontiers in education*, volume 5, page 129. Frontiers Media SA, 2020.

22. He He, Jordan Boyd-Graber, Kevin Kwok, and Hal Daumé, III. Opponent modeling in deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1804–1813, New York, New York, USA, 20–22 Jun 2016. PMLR.

Appendix

Name/Timestamp/Agent	Model	reward 1	reward 0	reward -1
3agents_linear/20230328-212250/best1	800	91.97	8.03	0.0
	900	88.21	9.86	1.93
	1050	92.24	7.02	0.74
	1150	91.34	8.06	0.6
	1200	92.38	7.35	0.27
	1250	90.86	9.07	0.06
	1400	91.14	8.56	0.3
agent_linear_decay099/20230327-185908/best1	5250	90.52	9.35	0.13
	5300	94.53	5.47	0.0
	5400	91.77	7.82	0.41
	5550	93.08	6.47	0.44
	5800	93.62	6.14	0.24
AgentTanH/20230327-192241/best1	250	92.97	6.92	0.11
	1150	91.94	7.35	0.71
	1300	89.88	9.73	0.39
best_agent_tiktaktoe_0998/20230327-191103/best1	3300	91.75	7.92	0.33
	3400	93.22	6.59	0.19
	3600	91.04	8.43	0.53
	3700	90.56	9.34	0.1

Table A1: Comparing the percentage of different rewards of trained DQN agents from different runs and iterations, that are trained to play Tic-Tac-Toe (-1 = losing, 0 = draw, 1 = winning).

Name/Timestamp/Agent	Model	reward 1	reward 0	reward -1
ConnectFour_linear/20230330-174353/best1	560	93.73	0.06	6.21
	720	94.0	0.02	5.98
	860	94.27	0.12	5.61
agent_ConnectFour_tanh/20230328-094318/best1	400	99.52	0.01	0.47
	550	99.07	0.05	0.88
	900	98.77	0.19	1.04

Table A2: Comparing the percentage of different rewards of trained DQN agents from different runs and iterations, that are trained to play Connect Four (-1 = losing, 0 = draw, 1 = winning).

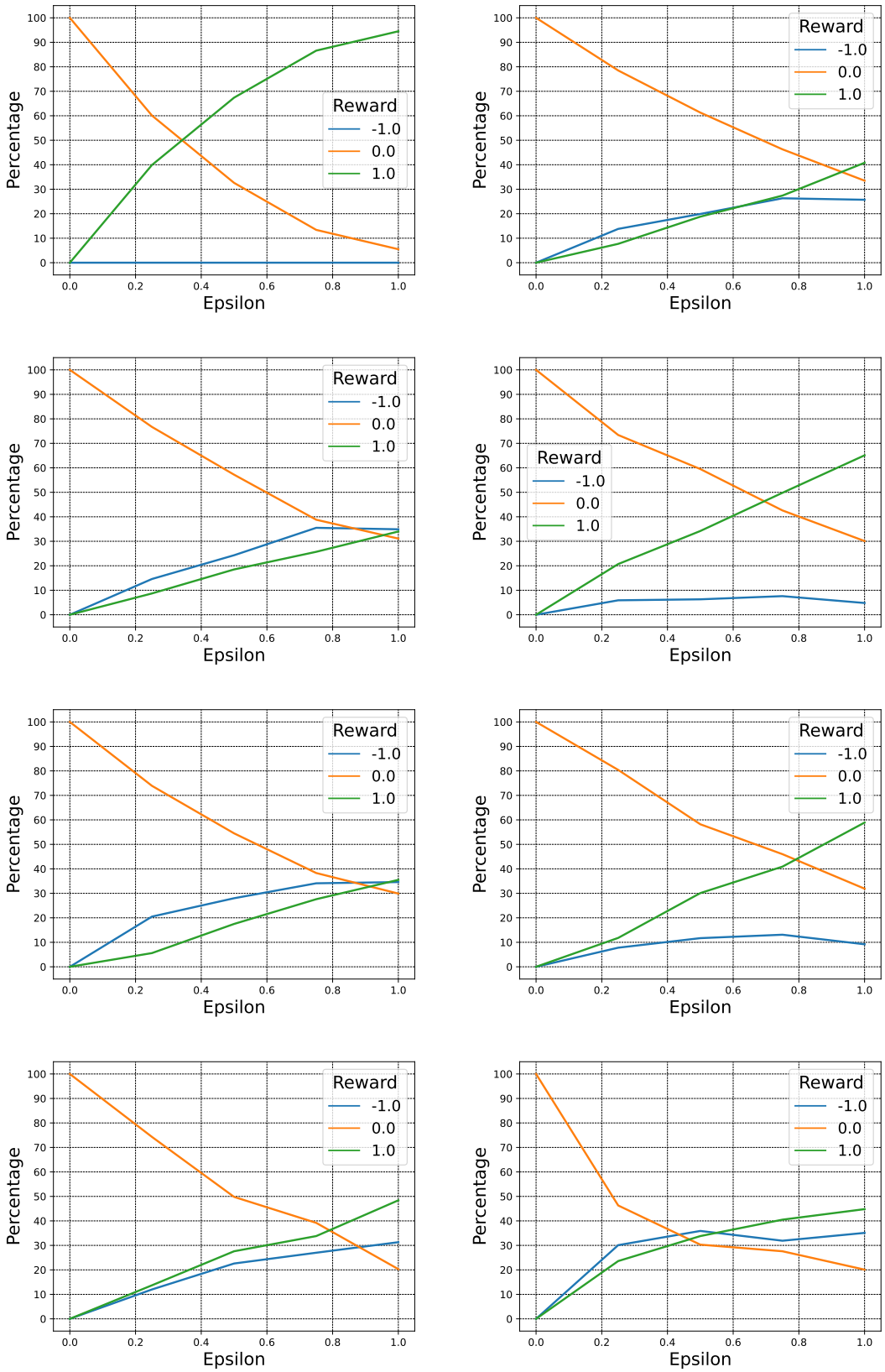

Figure A1: Showing the performance of different agents playing Tic-Tac-Toe against the best agent from section 4 using different ϵ values for the ϵ-greedy policy. (-1 = losing, 0 = draw, 1 = winning).
Top Left: High-performance DQN. Top Right: Adapting agent using Equation 1. Second Line: Adapting agent using Equation 3 on the Left and Equation 2 on the Right both with $c = 0.2$. Third Line: Adapting agents using Equation 4 with $B = -0.05$ on the Left and $B = 0.05$ on the Right. Bottom Left: Adapting agent using Equation 6. Bottom Right: Adapting agent using Equation 7 with $cp = 2$.

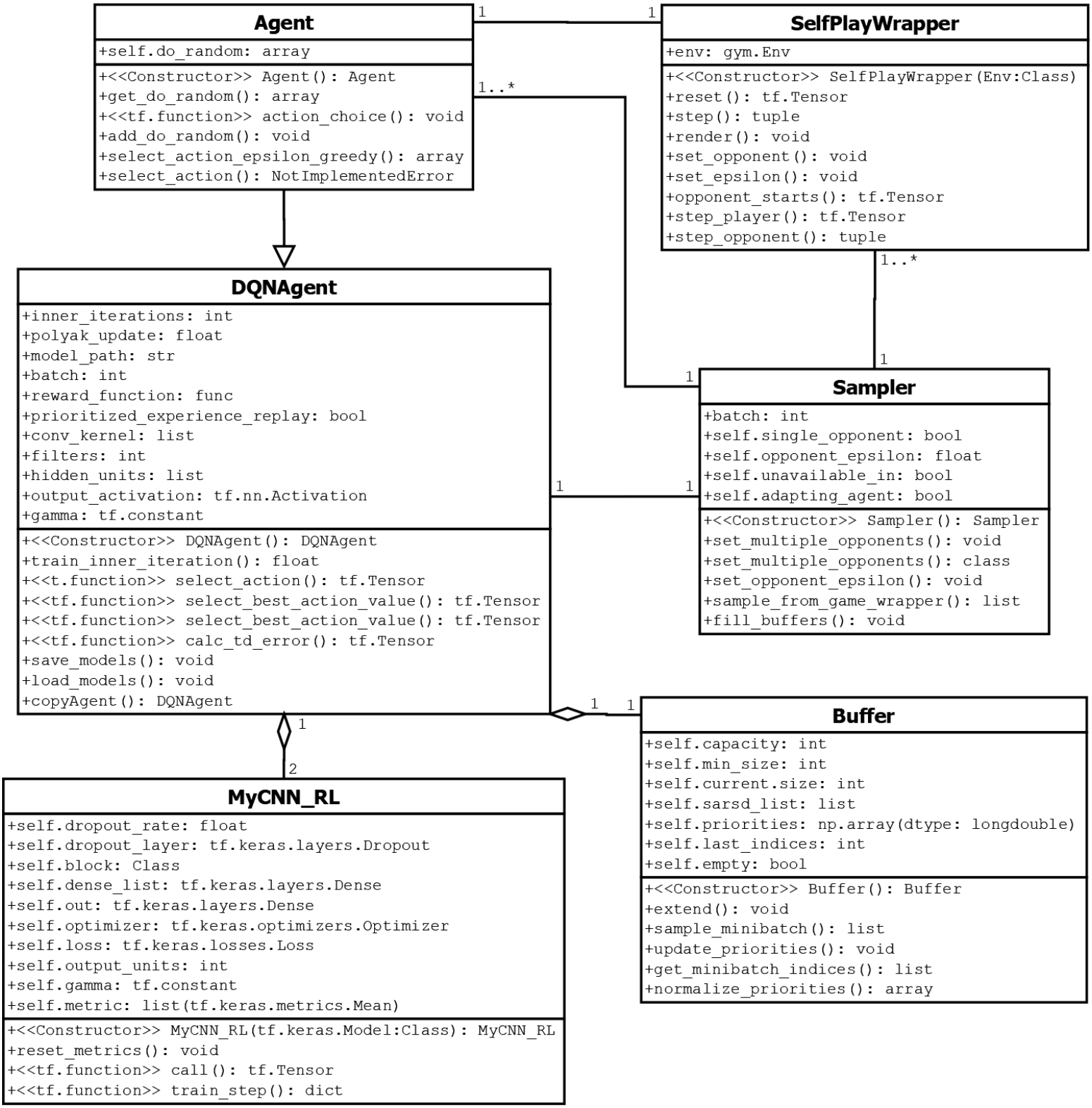

Figure A2: A class diagram of all classes concerned with the training of a DQN Agent.

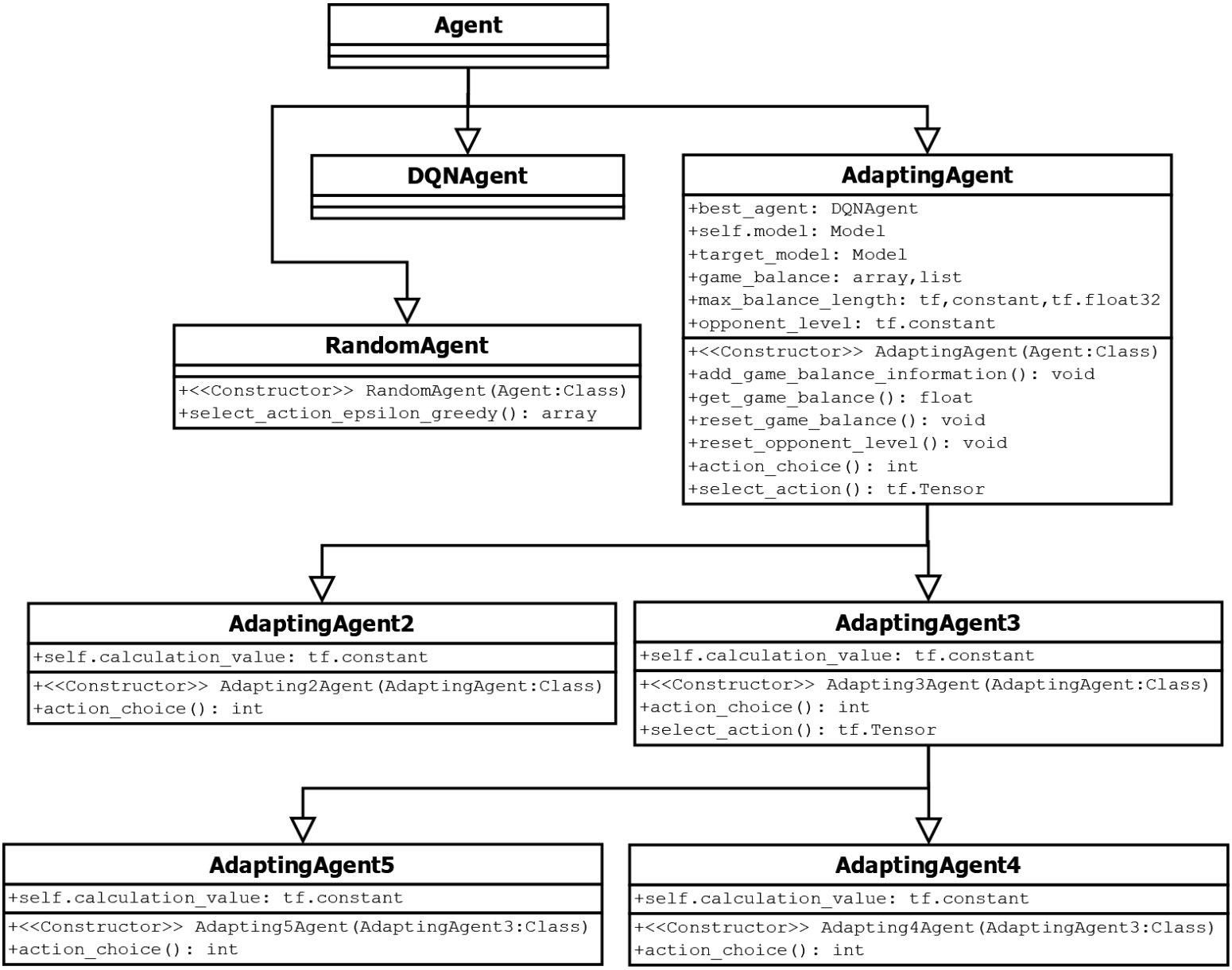

Figure A3: A class diagram of all Agent subclasses. The full information about Agent and DQNAgent can be found in Figure A2.