TRAINING A DEEP Q-NETWORK TO PLAY PONG

Philipp Widmann, s191818

https://github.com/PhilippWidmann/project-pong

ABSTRACT

One breakthrough technique in reinforcement learning was Deep Q-learning, which was first applied to training an agent to play Atari games. Since then numerous improvements have been made. This project implements a basic Deep Q-network (DQN) with extensions Double DQN and Prioritized Experience Replay and does experiments on the computationally feasible game Pong. Comparison then shows that all methods yield good results, but the extensions improve reliability of training. The code can be found at https://github.com/PhilippWidmann/project-pong.

***Index Terms*—** DQN, Double DQN, Prioritized Experience Replay, Pong

1. INTRODUCTION

One of the most influential papers of recent years in the field f reinforcement learning is [1]. It proposed the notion of a Deep Q-network to tackle reinforcement learning tasks and successfully applied it to the task of playing different Atari games. In this project, I strive to understand the concepts and implement them, as well as some improvements that were proposed since the publication of said paper. I will begin by giving an overview over the problem formulation and the theoretical basis of the algorithm before detailing some choices I made while implementing. Afterwards, I test the methods by applying them to the game Pong, which has the advantage of being solvable in reasonable time on readily accessible hardware.

2. MATHEMATICAL FOUNDATION

2.1. Basics of DQN

This subsection draws draw heavily on [1], particularly on the introduction and the discussion of methods.

The basic problem in reinforcement learning is the following: An agent observes a state $s_t \in S$ of the world and is presented with a selection A of possible actions. Each state-action pair is associated with a reward $r(s_t, a_t)$ and will result in a new state s_{t+1}, which may be unknown to the agent. Its goal is now to maximize a discounted future reward

$$R_t = \sum_{t=t'}^{\infty} \gamma r(s_t, a_t)$$

for a discount factor $\gamma \in (0, 1)$.

If we define π as the policy by which the agent chooses an action, this optimization problem starting in a state-action pair (s, a) can be solved by determining an optimal Q-function

$$Q^*(s, a) = \max_{\pi} \mathbb{E}[R_t \mid s_t = s, a_t = a, \pi],$$

and then always selecting the action with the highest value $Q^*(s, a)$.

However, determining this solution is in general infeasible. An approach to find an approximation to this optimal Q-function relies on the so-called Bellman equation

$$Q^*(s, a) = \mathbb{E}_{s'} \left[r + \gamma \max_{a'} Q^*(s', a') \mid s, a \right].$$

The idea is now to use this equation iteratively to update an estimate of the quality function, in the hope that this will converge to a good solution.

Specifically, in the context of a DQN we approximate the quality function $Q(s, a)$ by a deep neural network and use the difference in the Bellman equation as the loss function when doing training steps. In formulas:

$$y(s, a) = \begin{cases} r(s, a) \\ \text{if the episode ends at the next step;} \\ r(s, a) + \gamma \max_{a'} \hat{Q}(s', a'; \theta) \\ \text{else} \end{cases}$$

$$loss = (y_t(s, a) - \hat{Q}(s, a; \theta))^2,$$

where we denote the current neural net approximation of the Q-function as $\hat{Q}(s, a; \theta)$.

This alone is not enough to achieve convergence, however. The theoretical derivation relies on the inputs being independent. This can obviously not be fulfilled if the agent uses observations in the order in which they occur in the environment. Therefore, a replay buffer is introduced which stores a large number of observations and samples from these randomly, greatly reducing the dependence. In the basic version,

this sampling happens uniformly, but another method is discussed as an extension.

Furthermore, the iteration suffers from instability because in comparison to regular deep learning problems, the target values change in each step. To remedy this, one performs a split into a policy and a target network: The policy network is the one being trained. The target network is a copy of an earlier version of the policy network that is only periodically updated with a new copy of the policy network.

This is only a short overview of the necessary parts of a DQN. A deeper mathematical discussion can be found in [1], which also summarizes the complete procedure in [1, Algorithm 1].

2.2. Double DQN

An important observation about the instability of DQNs comes from [2]: Here it is shown that any kind of error (including estimation errors that naturally occur) will lead to an overestimation of the quality function. This is caused by the fact that the max-operator that determines the targets y_t is more likely to select an action that is overestimated (since it has a higher value than it should) and the resulting value will therefore also be overestimated. This can also lead to convergence to the wrong solution.

A remedy is presented in the form of Double DQN, where the max operator is split up into an argmax on the policy and an evaluation on the target network:

$$y(s,a) = \begin{cases} r(s,a) \\ \text{if the episode ends at the next step;} \\ r(s,a) + \gamma \hat{Q}(s', \arg\max_{a'} \hat{Q}(s', a'; \theta); \theta') \\ \text{else} \end{cases}$$

The hope is that overestimation in one network is somewhat uncorrelated from overestimation in the other and, thus, overall overestimation and its harmful effects are decreased.

2.3. Prioritized experience replay

Lastly, I use prioritized experience replay as proposed in [3]. The motivation is that not all observations are equally helpful in training the agent. Instead, one should focus on those observations that are "surprising" to the agent and replay these with a higher likelihood. Using the proportional variant proposed in the paper as a measure for this surprise, the probabilities of sampling an observation i are chosen according to

$$p_i \propto (y(i) - \hat{Q}(i; \theta))^\alpha$$

with constant $\alpha = 0.6$ as proposed in the paper and normalization constant such that the probabilities sum up to 1. New observations receive the maximum priority currently in the replay buffer, thus biasing the algorithm to look at each observation at least once. Note that I did not use any bias correction which is also developed in the paper.

3. IMPLEMENTATION DETAILS

3.1. Defining the environment

A working simulator for Pong is provided by the platform OpenAI gym [4]. This simulator accepts the actions "up", "noop" and "down" and returns a 210x160 pixel RGB frame together with rewards ± 1 when scoring or conceding a point. To introduce more randomness, every episode is started with a random number of up to 30 "noop" actions.

For performance reasons, this color image cannot be used directly as input. Instead I apply the same preprocessing steps that were established in [1]: The image is transformed into grayscale, the area containing the score is cut off and the rest scaled down into an 84x84 pixel image. Furthermore, the agent is only allowed to access every 4th frame – for intermediate frames, the last chosen action is simply repeated. This speeds up training by roughly times 4 while still allowing the agent enough flexibility and reaction time. Lastly, since the movement of the ball cannot be assessed from a still image, the last 4 observed frames are concatenated and served as a whole as input to the network.

I did not implement these preprocessing steps myself: The environment "PongDeterministic-v4" skips all but every 4th observation already. On top of that, I used (slightly modified) wrapper functions provided by the OpenAI baselines project [5].

3.2. DQN algorithm

The implementation of the DQN algortihm is done in Pytorch. It is a relatively straightforward realization of the algorithms briefly layed out beforehand. Apart from the preprocessing steps mentioned earlier, the code is my own work.

Of note may be the replay buffer, which I implemented directly on the GPU memory (provided a GPU is available for running the code). This way, in each operation only the one new observation must be transferred to the GPU instead of all 32 observations selected in the mini-batch. This resulted in a speedup of factor 2-3 on my local machine and on those machines provided by the DTU HPC cluster compared to my replay buffer in standard RAM. The inspiration for this was taken from [6], though the implementation is my own.

3.3. Network structure and hyperparameter choices

I use the same convolutional network defined in [1, Model architecture] and used as a quasi standard throughout literature when training on Atari games: The first three hidden layers are convolutional layers with 64, 32 and 32 channels, filter

sizes 8x8, 4x4 and 3x3 as well as strides 4, 2 and 1 respectively. Afterwards follows a fully connected hidden layer with 512 hidden units and ReLU activation functions. The output layer consists of 3 neurons corresponding to the 3 possible actions and is also fully connected.

The exploration-exploitation trade-off is handled by using an ϵ-greedy approach, annealing ϵ linearly from 1.0 to 0.01 over the first 100,000 iterations. Note that the replay buffer is prefilled with 50,000 observations (using randomly chosen actions) before training begins.

For training the network, an Adam optimizer with learning rate 0.0001 is used. Instead of the mean squared error loss described in the theoretical derivation, the Huber loss function is optimized to make training more robust against outliers. Finally, the number of training steps as well as the size of the replay memory can be dimensioned much smaller than in [1] since Pong is a much simpler task than many other Atari games. I found that the parameters proposed in the article [7], which also deals with the game of Pong exclusively, worked well for my purposes. A full list of hyperparameters can be found in Table 1.

Hyperparameter	Value
Reward discount γ	0.99
Training steps	1,000,000
Batch size	32
Target network update frequency	1,000
Learning rate (Adam)	0.0001
Replay buffer capacity	100,000
Replay buffer prefill	50,000
Exploration likelihood ϵ (initial)	1.0
Exploration likelihood ϵ (final)	0.01
ϵ decreases until frame ...	100,000

Table 1. Choice of hyperparmeters

4. NUMERICAL EXPERIMENTS AND DISCUSSION

Since I am mainly interested in improving the performance as much as possible, I chose to focus on three different scenarios: The basic DQN algorithm, the Double DQN algorithm and the Double DQN algorithm with prioritized experience replay. The reason for this order is that since Double DQN is a very easy, but beneficial extension, it is employed in almost all scenarios. It would, therefore, not make sense to look at a basic DQN implementation with prioritized experience replay.

In order to compare the implementations, I run the training algorithm 10 times each and protocol the results. Since one game episode lasts until one player reaches 21 points and the reward per game episode is the final score of the agent minus the final score of the other player, the reward per game episode can range from -21 (very bad) to +21 (very good).

4.1. Training performance

At first, consider the quality of the agent during training, which is depicted in Figure 1. As we hoped, we observe a similar picture for all three scenarios: The reward begins at or close to -21 when the agent is effectively choosing random actions and increases up to or close to +21 over the course of training. Furthermore, once the reward reaches an almost perfect score, it tends to fluctuate slightly but not decrease significantly again – regardless of which algorithm is used.

Nevertheless, there are also some significant differences: The first thing to note (which may be somewhat hard to immediately tell from the graph) is that in the case of basic DQN, 3 different runs do not converge to a successful solution, but instead deliver a solution that produces a reward of -21. Compared to that, only 1 run using Double DQN fails and none when also using prioritized experience replay.

On top of that, experience replay also seems to have a marked effect on the speed of convergence. While basic and Double DQN both achieve close to optimal results after about 450,000 iterations for most of the runs, adding prioritized experience replay reduces that threshold to about 350,000 frames. And lastly, prioritized experience replay also shows fewer outliers and might have overall reduced variance (though a larger test data would be necessary to support this last claim).

Overall, these results imply that the extensions to DQN and in particular prioritized experience replay greatly improve the stability of the algorithm. I would hypothesize that this is caused by the reward structure of the Pong environment: An episode takes around 900 frames in the beginning for a very one-sided game and it can take up to 10,000 frames or more in the middle of training. However, only a maximum of 41 of these frames actually carries a non-zero reward! This very sparse reward structure means that that successful training depends on making "lucky" observations and *also training on them*. That is exactly the scenario that prioritized experience replay was designed for (cf. [3, 3.1]): Choosing those observations that have the most effect on training. Consequently, these results should not be surprising.

4.2. Final network performance

The second interesting metric for the success of the algorithms is the performance of the agent after training. To examine this, each trained agent is run for 50 episodes while always choosing the best action according to the trained network. The average reward per episode is given in Table 2 for each agent.

First off, we notice that obviously the final performances align with the final training performances (this is unsurprising since training only includes random actions with probability 1%). A (suspiciously) large amount of agents even manages to achieve perfect scores. However, some runs like the DDQN run scoring 11.96 or the DDQN run with prioritized experi-

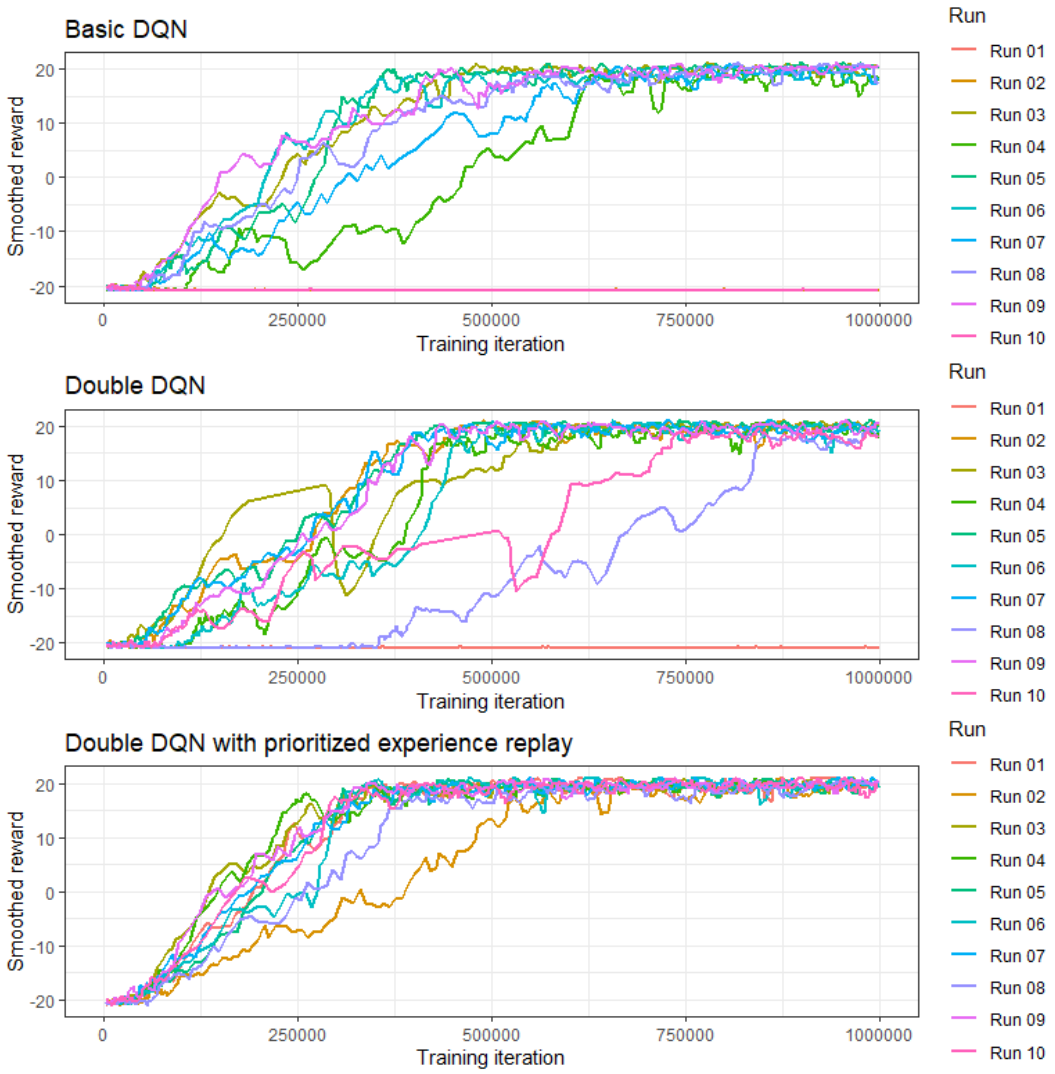

Fig. 1. Training rewards per episode for the three considered scenarios over 10 test runs. The rewards are smoothed over the last 5 data points to make the graphs more easily readable. Note that in the case of basic DQN 3 runs stay at a negative reward of -21, compared to only 1 in the case of DDQN and none when priority replay is also included.

Algorithm	Mean test rewards per agent
Basic DQN	-21.0 (x3), 18.44, 21.0 (x6)
DDQN	-21, 11.76, 17.96, 20.92, 21.0 (x6)
DDQN & prio. replay	-3.36, 16.0, 21.0 (x8)

Table 2. Mean test rewards of all trained agents over 50 test runs each

ence replay scoring only -3.36 perform significantly worse than expected. Taking a closer look at these outliers reveals that these agents manage to score either +21 or -21, but nothing in between.

This points to a structural problem in our problem environment: If we observe a fully trained agent during play (as can be done in the GitHub repository), we notice that it has perfected exactly one move while essentially overfitting and "unlearning" normal playing behaviour. This move allows it to score points consistently and almost immediately after the ball respawns, which works due to the very limited randomness in the simulator. If an unfortunate update of the network during the final stages of training or a rare simulation prevents the agent from successfully executing this move, however, he cannot recover.

One immediate solution to this problem would be to stop training earlier: We see from Figure 1 that training for most runs could be ended after around 500,000 iterations, leaving less time for the network to overfit. An even better solution would include a simulator with a higher degree of randomness, but that is beyond what I am able to achieve in this

project.

5. CONCLUSION

While the basic Deep Q-network is already sufficiently powerful to solve the game of Pong, extending it to a Double DQN and especially including prioritized experience replay can greatly increase the stability and speed of training. Of course, these rather simple extensions are not the end of the line. The more recent paper [8] successfully combines six different extensions to the DQN algorithm, achieving vast performance improvements. Taking a closer look and understanding these methods would be an interesting endeavour for the future.

However, when doing so it would most likely be useful to switch over to a more challenging task than Pong, since the methods we explored in this project already solve the game almost optimally. Given that the absolute score is capped at +21, any improvements made by additional extensions would become increasingly hard to measure.

6. REFERENCES

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. Bellemare, A. Graves, M. Riedmiller, A. Fidjeland, G. Ostrovski, et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[2] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI Press, 2016, pp. 2094–2100.

[3] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[4] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[5] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov, "Openai baselines," https://github.com/openai/baselines, 2017.

[6] B. Parr, "Deep in-GPU experience replay," *CoRR*, vol. abs/1801.03138, 2018.

[7] M. Lapan, "Speeding up DQN on PyTorch: how to solve Pong in 30 minutes," https://medium.com/@shmuma/speeding-up-dqn-on-pytorch-solving-pong-in-30-minutes-81a1bd2dff55, 2017.

[8] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.