1 Introduction

1.1 Reinforcement Learning

In reinforcement learning, sequential decision making problems need to be solved by an agent. The agent is provided with some input state and then needs to choose an action. Input state and chosen action are used to create a subsequent state and a reward signal for the agent. By maximizing the reward signals the agent develops a policy for acting in the environment and thus gets better in solving the given problem. Finding the optimal policy is the goal of reinforcement learning. Hereby the environment is fully characterized by the state.
Time is discrete in reinforcement learning. Therefore the consecutive states form a chain. Nevertheless transitions from one state to its subsequent state are not dependent on history, they only depend on the current state. This is called the Markov assumption.
For finding the optimal policy the state value function $V_\pi(s)$ can be used. It is defined as the expected reward of a policy π that can be reached by starting at the state s

$$V_\pi(s) = \mathbb{E}\left[R\right]|_{\pi,s}. \tag{1}$$

The expected reward R for a given state s_0 is defined as the sum of the current reward and the discounted future rewards

$$R = \sum_{t=0}^{\infty} \gamma^t R_t. \tag{2}$$

$\gamma \, \epsilon \, [0, 1]$ is the discount factor.
The optimal policy π^* maximizes the state-value function.

1.2 Neural Networks

2 DQN

2.1 Q-learning

In Q-learning [2] instead of the state-values, Q-values are being used. They are defined as the current reward depending on the chosen action and the

discounted future rewards under the premise of always using the policy π for further decision making after the initial action choice:

$$Q_\pi(s,a) = \mathbb{E}\left[R\right]|_{a,s,\pi}. \tag{3}$$

The best policy is then determined by maximizing Q_π over the actions in every step. Usually it is not possible to learn every action for every state explicitly because the problems are often to extensive. Therefore the Q-values get approximated by a parametrized function $Q(s,a,\theta_t)$ with θ_t being the parameters. The Q-function is then updated towards the target value

$$Y_t^{\mathrm{Q}} \equiv R_{t+1} + \gamma \max_a Q\left(s_{t+1}, a, \theta_t\right) \tag{4}$$

after each action via:

$$\theta_{t+1} = \theta_t + \alpha\left(Y_t^{\mathrm{Q}} - Q\left(s_t, a; \theta_t\right)\right)\nabla_{\theta_t} Q\left(s_t, a; \theta_t\right). \tag{5}$$

This makes it an off-policy algorithm, as the optimal Q-value: Q^* is approximated by Q directly (regardless of the followed policy) in contrast to for example SARSA. The policy is still important as it determines, which state-action pairs are used to update the model [5]. Q-learning is also a model free approach, as it doesn't make a model of the environment but instead directly estimates Q^*.

2.2 Q-Networks

For some reinforcement learning problems, the simplest implementation of the Q-learning algorithm, a table is perfectly sufficient to find a good policy. If we consider more complicated problems with bigger inputs other methods are needed. For state of the art reinforcement learning, usually neural networks are being used. They are especially convenient as they are trained from raw inputs, which makes handcrafted features redundant [3]. Neural Networks which are implemented to learn with the Q-algorithm are called Q-networks. They are non linear function approximations for $Q(s,a,\theta_t)$. To update the Q-network the loss function

$$L_t\left(\theta_t\right) = \mathbb{E}_{s,a\sim\rho(s,a)}\left[\left(Y_t^{\mathrm{Q}} - Q\left(s,a;\theta_t\right)\right)^2\right] \tag{6}$$

is used, where $\rho(s, a)$ is a probability distribution over states and actions. To make this compatible with the Q-learning algorithm, the weights need to be updated at every step and the expectations exchanged with samples from the probability distribution $\rho(s, a)$.

2.3 DQN

Deep Q-networks [3] are multilayered neural networks which make use of the Q-learning algorithm. The two mayor improvements of the model are the use of two separate networks and an experience buffer. The two networks are called online network and target network. The target network is used to calculate the targets Y_t^{DQN}. It has the same structure as the online network but to make learning more stable, the weights of the target network θ_t^- stay constant for a longer time. They are being copied from the online network every τ steps. The target is then calculated by:

$$Y_t^{\mathrm{DQN}} \equiv R_{t+1} + \gamma \max_a Q\left(s_{t+1}, a; \theta_t^-\right). \tag{7}$$

If only one network was used, an update of $Q(s, a)$ would often not only lead to a higher value of $Q(s_t, a_t)$ but also to higher expected Q-values $Q(s_{t+1}, a)$ for all actions. If the target was also calculated by this network this could then lead to oscillations or divergence of the policy [3]. The additional improvement of DQN is experience replay. Without experience replay, only new experiences are used in training and discarded right afterwards. Therefore important but rare experiences are almost immediately forgotten and the updates are not independent and identical distributed but strongly correlated. To address this problem, an experience buffer is implemented. There the experiences are stored and then at training time sampled uniformly at random. Usually a simple FIFO algorithm is being used. But there are more sophisticated methods for this, one of those is discussed later.

3 Rainbow / DQN Extensions

3.1 Double DQN

Q-learning and DQN tend to learn overestimated action values. During the maximization over the action choices, values are rather over than underestimated. This isn't necessarily a problem if they are uniformly overestimated

or if interesting experiences are overestimated. However, in DQN overestimations differ for different actions and states. Combined with bootstrapping, this results in the propagation of wrong values and thus to worse policies [2]. To reduce those overestimations, the Double Q-learning algorithm [2] is used. The main idea of the Double Q-learning algorithm is to decouple value selection and evaluation. DQN, with separate online and target networks provides an excellent framework for this decoupling: The online network is used to select an action from the action choices via maximization whereas the target network evaluates the actions to generate the Q-values. The resulting double DQN yields more accurate values and hence leads to better policies than DQN [2]. Updating double DQNs is similar to updating DQNs when the target is rewritten as:

$$Y_t^{\text{Double DQN}} \equiv R_{t+1} + \gamma Q\left(S_{t+1}, \underset{a}{\text{argmax}} Q\left(S_{t+1}, a; \theta_t\right), \theta_t^-\right). \tag{8}$$

θ_t^- and θ_t are the weights of the target and online network respectively.

3.2 Prioritized Experience Replay

In standard experience replay, the agent is forced to pick experiences uniformly from all experiences in its memory. Therefore all experiences are sampled with the same frequency that they were originally encountered. This is not necessarily good for the learning process, as some experiences might not hold any valuable information for the agent but occur very often while other rare situations could be crucial for learning.
This can be improved by prioritized experience replay [4]. Here every experience in the buffer gets a priority according to its TD-error. The TD-error measures the difference between the actual Q-value and the Target-Q-value, so if experiences with bigger TD-errors are provided with bigger priorities, experiences from which there still is a lot to learn are favoured. The priority p_i is determined from the TD-error δ_i according to:

$$p_i = |\delta_i| + \epsilon. \tag{9}$$

Hereby $\epsilon > 0$ denotes a small parameter to ensure, that every experience has a priority bigger than zero and thus can be picked for a sample batch. New experiences are always added with maximum priority to the memory. Problematic with this greedy approach is, that only experiences that are

picked for learning get their priority updated. Therefore experiences with low initial priority might, because of the buffer memory structure be removed from memory before they could have been picked for learning. The buffer is also very sensitive to noise spikes [4].
To overcome this problems stochastic prioritization is used and p_i adjusted according to:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}. \tag{10}$$

Here $\alpha \, \epsilon \, [0, 1]$ is another parameter which adjusts the amount of prioritization that is used. For $\alpha = 0$ we get the uniform case (no prioritization), whereas $\alpha = 1$ leads to greedy prioritization.
Stochastic prioritization introduces bias to our model. This needs to be considered for updating the model, because it could change the solution the model is converging to [4]. To correct this, importance sampling weights (IS weights) are introduced:

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)} \right)^\beta, \tag{11}$$

with N being the replay buffer size and $\beta \, \epsilon \, [0, 1]$ being a hyperparameter for adjustment of the bias. For $\beta = 1$ the bias gets fully compensated. This is most important at the end of the training process. Therefore β starts at an initial value and is then being annealed during training.

An efficient data structure for the memory is crucial for good performance. To guarantee this, we implemented a sum tree to store the data, where searching is of complexity $O(1)$ and updating of complexity $O(\log N)$. A sum tree is a binary tree, in which the parent node values are the sum of the child node values. In our sum tree, the transition priorities were saved in the leave nodes. Therefore the root holds the total priority. An array was used to hold the associated data values to the priorities. For the purpose of sampling the total priority is divided into k priority ranges, with k being the number of experiences in one sample. From each of these priority ranges one value is sampled uniformly and its corresponding leave node is searched. The data belonging to this priority is than used for the sample.

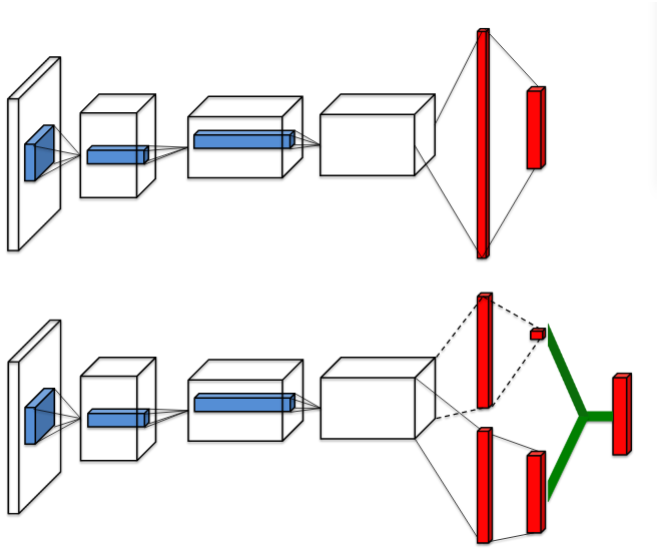

Figure 1: Top: Standard Deep Q-network. Bottom: Dueling DQN with two separate streams [6].

3.3 Duelling networks

The idea behind duelling networks [6] is, that for some states only the state-value function is important while for others the chosen action is crucial.
Consider a small toy problem: Our agent needs to catch coins which are falling down form above, he can either move right or left or wait to catch them. In some states there are no coins at all. In that state it is not important which action is chosen, whereas for other states its is. To exploit this the neural network is split into two streams: the action and the state-value stream, as can be seen in figure 1. The network is split after the convolutional layers. Therefore the two streams consist of linear layers. The benefit of this is, that it is possible to get separate estimations for state-value function and action-value function. Hereby the state-value $V(s;\theta,\beta)$ is a scalar property while the action-vector $A(s,a;\theta,\alpha)$ has the dimension of the quantity of possible action choices, in our case six. θ are the weights of the convolutional layers, while α and β are the weights for the A- and the V- stream respectively.
For getting the Q-value A and V need to be recombined. It is not a good

idea though to simply add them together:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + A(s, a; \theta, \alpha). \tag{12}$$

In that case it would get impossible to retrieve V and A from Q uniquely. One could add a constant to A and subtract it from V without changing Q. It is better to calculate Q by

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + \left(A(s, a; \theta, \alpha) - \frac{1}{|\mathcal{A}|} \sum_{a'} A\left(s, a'; \theta, \alpha\right) \right), \tag{13}$$

where A and V keep their identifiability and the optimization becomes more stable [6].

3.4 Noisy Networks

So far we did use an ϵ-greedy policy for exploration. Another technique which has been found to produce better result for many of the Atari games are Noisy Nets [1]. They add parametric noise to the weights and thus aid exploration without the need to pick random actions as part of a policy (e.g. ϵ-greedy). This is very convenient, because there is no need to tune additional hyper parameters as the reinforcement learning algorithm tunes the weights automatically. Consider a neural network $y = f_{\theta_n}(x)$. Where θ_n are the noisy weights. A linear layer of a neural network can be written as:

$$y = wx + b, \tag{14}$$

whereas a noisy linear layer can be written as:

$$y = \left(\mu^w + \sigma^w \odot \varepsilon^w\right) x + \mu^b + \sigma^b \odot \varepsilon^b. \tag{15}$$

Here x is the input, w, and $\mu^w + \sigma^w \odot \varepsilon^w$ are the weights and b and $\mu^b + \sigma^b \odot \varepsilon^b$ are the biases for linear and noisy linear layer respectively. All of the named parameters are trainable except for ε^w and ε^b which are noise random variables. We chose factorized gaussian noise for the distribution of the ε parameters, as it reduces computation time for random number generation, which is important for single thread-agents such as ours [1]. Here only one independent noise per input and another independent noise per output is needed, in contrast to independent Gaussian noise, where one independent

noise per weight would be required. We factorized ε^w to $\varepsilon^w_{i,j}$. The noise random variables can then be written as:

$$\begin{matrix} \varepsilon^w_{i,j} = f\left(\varepsilon_i\right) f\left(\varepsilon_j\right) \\ \varepsilon^b_j = f\left(\varepsilon_j\right) \end{matrix}, \tag{16}$$

where we used

$$f(x) = \text{sgn}(x)\sqrt{|x|}. \tag{17}$$

The parameters $\mu_{i,j}$ were initialized as samples from a random uniform distribution $\left[-\frac{1}{\sqrt{p}}, +\frac{1}{\sqrt{p}}\right]$ with p being the number of inputs for the noisy linear layer. $\sigma_{i,j}$ were set as $\sigma_{i,j} = \frac{\sigma_0}{\sqrt{p}}$.
For the Noisy Networks implementation we replaced the Fully Dense layers of the state-value and action streams by Noisy layers. [1]

3.5 Multi-Step Learning

We are trying to learn the total discounted return

$$Q^*(x_t, a_t) = R(x_t, a_t) + \gamma R(x_{t+1}, a_{t+1}) + \gamma^2 R(x_{t+2}, a_{t+2}) + ...$$

If we assume that we use the optimal policy for $t+1$ onward, we can substitute $\gamma R(x_{t+1}, a_{t+1}) + \gamma^2 R(x_{t+2}, a_{t+2}) + ...$ with $Q^*(x_{t+1}, a*_{t+1}$.

Q-Learning takes the reward from a single step and adds the discounted value estimate of the greedy action at the next step.

$$\hat{Q}^*(x_t, a_t) = R(x_t, a_t) + \gamma \hat{Q}^*(x_{t+1}, a_{t+1})$$

For multi-step Q-Learning we take the n-step truncated return

$$R^{(n)}_t \equiv \sum_{k=0}^{n-1} \gamma^{(k)}_t R_{t+k+1}$$

to estimate the

4 Other things

4.1 Augmented data

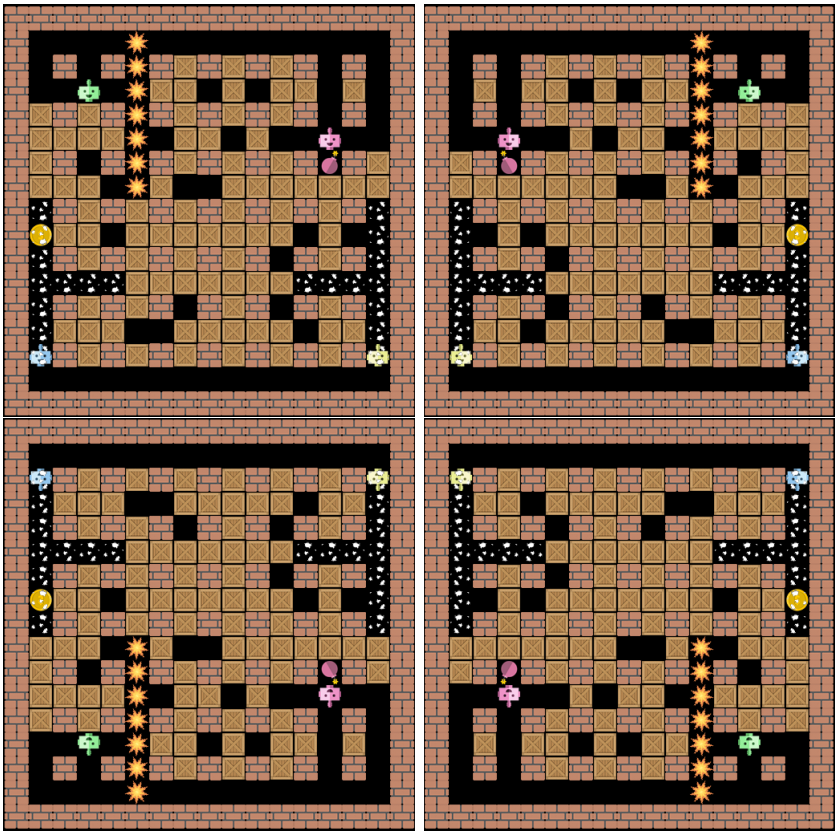

Figure 2: Data augmentation:
Upper left: original; Upper right: horizontal mirroring;
Lower left: vertical mirroring; Lower right: combined mirroring

As the inputs for our bomberchamp agent are symmetric, we wanted to use data augmentation to increase the number of samples for training and to make learning more symmetric. From each original sample, three augmented samples were created. For augmentation, we had to mirror the environment

and to change action choices accordingly. The augmented environment consisted of horizontal mirroring, vertical mirroring and a combination of both as seen in figure 2. For horizontal mirroring the agent choices left and right were exchanged, for vertical mirroring up and down and for the combination both were swapped.
Data augmentation did not work as well as we expected, so it was not included in our final implementation (more details in the next section).

4.2 Centring of agent

4.3 Invalid actions

4.4 Auxilliary Reward Design

4.5 Minigame Collection

4.6 Self-Play (maybe)

5 Observation

5.1 It does not work

5.2 Network scaling

6 Summary and Improvements

7 Model

8 Training process

8.1 Centring of the agent

To simplify learning for our agent, it was centred on the board, so that it stays at a fixed position while the environment (coins, crates, other agents) move around it. This is very common in the Atari games on which most implementations of DQN agents are tested, including the Rainbow agent without distributional reinforcement learning from which we took most of our initial hyper parameters. Therefore we thought, a more similar setting

would be beneficial to our agent. To implement this, the size of the board was increased to four times its original size and the agent placed in the middle of the new board.

8.2 Self play

For training purposes a simple agent implementation that follows the rules of bomberman and plays reasonably well was provided. To exploit this we wanted to first train our agent by classifying inputs generated with the simple agents. This is why we implemented a self play strategy. This was also useful, as we could then use google colab and train more than one agent at the same time while using the same neural network.

References

[1] Meire Fortunato et al. "Noisy Networks for Exploration". In: *CoRR* abs/1706.10295 (2017). arXiv: 1706.10295. URL: http://arxiv.org/abs/1706.10295.

[2] Hado van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-learning". In: *CoRR* abs/1509.06461 (2015). arXiv: 1509.06461. URL: http://arxiv.org/abs/1509.06461.

[3] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

[4] Tom Schaul et al. "Prioritized Experience Replay". In: *CoRR* abs/1511.05952 (2015). arXiv: 1511.05952. URL: http://arxiv.org/abs/1511.05952.

[5] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. 1st. Cambridge, MA, USA: MIT Press, 1998. ISBN: 0262193981.

[6] Ziyu Wang, Nando de Freitas, and Marc Lanctot. "Dueling Network Architectures for Deep Reinforcement Learning". In: *CoRR* abs/1511.06581 (2015). arXiv: 1511.06581. URL: http://arxiv.org/abs/1511.06581.