Heidelberg University
Institute of Computer Science

Project report for the lecture Fundamentals of Machine Learning

Reinforcement Learning for Bomberman

`https://github.com/nilskre/bomberman_rl`

Team Member: Felix Hausberger, 3661293,
Applied Computer Science
eb260@stud.uni-heidelberg.de

Team Member: Nils Krehl, 3664130,
Applied Computer Science
pu268@stud.uni-heidelberg.de

Abstract

Plagiarism statement

We certify that this report is our own work, based on our personal study and/or research and that we have acknowledged all material and sources used in its preparation, whether they be books, articles, reports, lecture notes, and any other kind of document, electronic or personal communication.

We also certify that this report has not previously been submitted for assessment in any other unit, except where specific permission has been granted from all unit coordinators involved, or at any other time in this unit, and that we have not copied in part or whole or otherwise plagiarized the work of other students and/or persons.

Member contributions

Nils Krehl

tbd

Felix Hausberger

First, literature research was done on Q-learning itself, going over to approximative Q-learning, Deep Q-learning, double DQNs and dueling DQNs. Then further improvements were investigated by looking at the prioritized experience replay buffer concept, different exploration methods and the rainbow paper from Google DeepMind. Also Bomberman related papers and experience reports for Reinforcement Learning were analyzed, also for hyperparameter tuning.

Then the Q-learning algorithm got implemented using both an online and target network. Responsibility was also taken for both the normal and the prioritized experience replay buffer including the SumTree data structure. After shaping the rewards, choosing the right exploration method and setting the hyperparameters based on the research results, the training of the agents was executed, visualized and analyzed. Possible improvements for the problems during the training phase were considered and applied. The reasons for the bad local optimum were analyzed. The experimental results, open problems and improvements were then summarized in the conclusion.

Contents

List of Abbreviations

CUDA	Compute Unified Device Architecture
DQN	Deep Q-Networks
ELU	Exponential Linear Unit
ICAART	International Conference on Agents and Artificial Intelligence
MDP	Markov Decision Process
ReLU	Rectifier Linear Unit
PER	Prioritized Experience Replay
PPO	Proximal Policy Optimization

0 Project Setup

For a detailed description of how to set up the project, please have a look at `https://github.com/nilskre/bomberman_rl/blob/master/README.md`.

1 Introduction

Reinforcement Learning is a part of Machine Learning, where an agent is trained to interact in a desired way with its environment. Based on the current state, the agent decides for an action and can receive a reward for the chosen action. [17]

The potential of Reinforcement Learning is proven many times in varying contexts. E.g. attention was generated by the success of DeepMind's AlphaGo, the first artificial agent defeating a human in the game Go. For training this agent they used Reinforcement Learning. [19]

Salvador, Oliveira and Breternitz have summarized the evolution of Reinforcement in a literature review [17]. They start in 1989 with the publication introducing Q-learning. In the recent past many publications deal with the combination of Deep Learning with Reinforcement Learning. This area is known as Deep Q-learning.

As part of this project Reinforcement Learning is used for learning how to play Bomberman. Bomberman is a strategic board game, which is played on a field containing walls, crates, bombs, coins and other players. For winning the game one must gather points by killing other players and collecting coins. Typically in each game round first the walls around the player are removed by placing bombs. Next the agent can navigate towards coins or towards his opponents and kill them by placing bombs. [15]

After introducing the fundamentals and related work in chapter 2, the approach is described in chapter 3. Therefore the selected Reinforcement Learning method and the training process are presented. In chapter 4 the results of the experiments are described. A conclusion is drawn in chapter 5.

2 Fundamentals and Related Work

Q-learning is a known off-policy and model-free approach to train an agent based on temporal difference in an environment that can be modeled as a Markov Decision Process (MDP). An agent therefore does not necessarily use the policy it is trained for and does not know the transition probabilities and rewards in the MDP beforehand. Equation 1 shows the iterative update

formula for the Q-values that an online model uses to choose the right action [8].

$$Q_{k+1}(s,a) = (1-\alpha)Q_k(s,a) + \alpha(r + \gamma \max_{a'} Q_k(s',a')) \tag{1}$$

The problem with conventional Q-learning is that in most of the cases the state dimension is far too high to explore and model the MDP entirely in foreseeable future. To deal with this problem the Q-values need to be approximated using a regression model. Deep neural networks have proven to be highly applicable for this task, which leads to the term of *Deep Q-learning* and respectively *Deep Q-Networks* (DQN) for such network architectures. DQNs use the vectorized numerical state as its input and outputs the predicted Q-values. It learns through backpropagating the temporal difference error over each step for a single neuron. Note that for the Q-learning algorithm a backpropagation is done after every step of the simulation. To avoid temporal correlation between succeeding experiences an experience replay buffer is used to randomly sample a training batch in each step. Also rare experiences will be used more frequently to update the model parameters using this approach. In the following papers regarding DQN architectures shall be introduced as well as papers dealing with the bomberman environment for Reinforcement Learning.

Paper [10] tackles the problem that the *max* operator in Equation 1 often leads to overoptimistic value estimates as the DQN uses the same Q-values to both select and evaluate an action. It therefore changes the iterative update formula to 2.

$$Q_{k+1}(s,a) = (1-\alpha)Q_k(s,a) + \alpha(r + \gamma Q'_k(s', arg\max_{a'} Q_k(s',a'))) \tag{2}$$

Now a target DQN is used to separate the determination of the greedy policy, which is still done by the online network, from the Q-value estimation. Using this double DQN approach results in less overestimated Q-values and therefore better policies by more accurate Q-value estimates. It also makes the learning process more stable and reliable. The weights are copied from the online network to the target network after a fixed amount of episodes.

Another optimization was introduced in paper [20]. It changes the architecture of the DQN by splitting it into two separate value streams. One stream estimates the state value function and the other one the state-dependent action advantage function. Both streams are then combined again using a special aggregating layer to produce an estimate of the Q-values (see Figure 1).

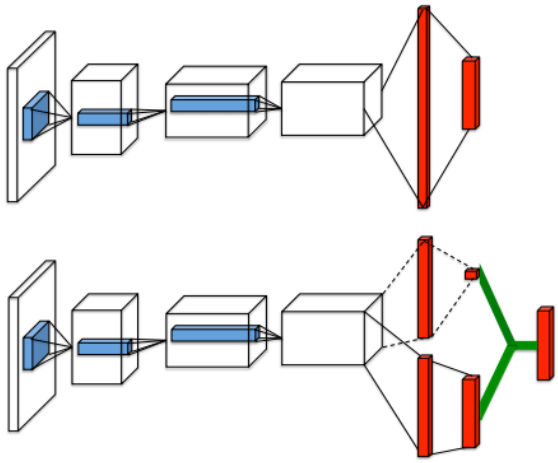


Figure 1: Architecture of a normal DQN (top) compared to a dueling DQN (bottom)

The state-dependent action advantage function is defined as

$$A^{\pi}(s,a) = A^{\pi}(s,a) - V^{\pi}(s) \tag{3}$$

and measures the importance of each action. The special aggregating layer uses Equation 4 to estimate the Q-values while also tackling the issue of identifiability.

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + (A(s,a;\theta,\alpha) - \frac{1}{|A|}\sum_{a'} A(s,a';\theta,\alpha)) \tag{4}$$

Using the dueling architecture approach an agent can learn which states are valuable independently of each action, which is especially useful in case actions do not influence the environment in any useful way. This leads to a better and more robust policy evaluation in the presence of many similar-valued actions. Also when looking at the last layer of a conventional DQN (see Figure 1) it usually becomes much more sparse and biased. As the state value is modeled as a single neuron in the dueling architecture, learning the state value function becomes much more efficient.

Obviously, approaches exist that combine double Deep Q-learning with dueling architectures like [13].

Besides optimizing the learning process itself and the model architecture, [18] now proposes a way to sample more efficiently from the experience replay

buffer. Each experience is assigned a learning priority score p_i with

$$p_i = \frac{1}{rank(i)} \tag{5}$$

where $rank(i)$ is the rank of experience i in the priority queue built upon the magnitude of the temporal difference error δ of each experience. The stochastic sampling probability of each experience is then calculated according to

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \quad with\ 0 \leq \alpha \leq 1. \tag{6}$$

The hyperparameter α determines the degree of using prioritization over random sampling. Using this stochastic sampling approach tackles the problem of a diversity loss and subsequent over-fitting when just greedyly sampling according to the magnitude of δ. One could also choose

$$p_i = |\delta_i| + \epsilon \tag{7}$$

instead of 5 but the latter is more prone to outliers. Furthermore, each gradient descent step during backpropagation needs to be weighted with

$$w_i = (\frac{1}{N}\frac{1}{P(i)})^\beta \tag{8}$$

to counter a bias towards high prioritized experiences introduced using the prioritized experience replay buffer. The weights should also be normalized with $\frac{1}{max_i w_i}$. As an unbiased nature of weight updates is especially important during the last training updates near convergence, β increases slowly from a start value β_0 to 1 over time.

There are several other improvements mentioned in the rainbow paper [11] like multi-step learning, distributional Reinforcement Learning and noisy nets, but these are out of the scope for this small research project.

Other papers were found that explicitly use DQNs within the Bomberman environment. One of them being [15] which introduces two novel exploration strategies, Error-Driven-ϵ and Interval-Q, and compares them to conventional exploration strategies like Diminishing ϵ-Greedy and Max-Boltzmann, whereas Max-Boltzmann with decreasing temperature parameter still performs best in the long run by empirical evaluation. Nevertheless Error-Driven-ϵ, despite being less stable, learns faster than all other exploration techniques. The paper also gives an approach to encode the state. Therefore, the playing field is encoded by four matrices, which are computed as:

- Free, breakable and obstructed cells are encoded as either 1, 0 or -1,

- The position of the player is encoded as 1, free cells as 0,
- The position of opponent players are also encoded as 1, free cells as 0 and
- The danger score for each cell is calculated as $\frac{timepassed}{timeneededtoexplode}$, which gets an additional negative sign in case a bomb was planted by the player itself.

The paper also gives valuable insights into the configuration of hyperparameters, rewards and the amount of training needed until convergence, which is about 100 generations à 10.000 episodes.

[7] uses an imitation-based learner that trains its model with the actor-critic proximal-policy optimization method in the Bomberman environment. Here insights about how rewards need to be chosen and which state representation to choose can also be derived.

Bomberman seems to provide a perfect environment to try out different Reinforcement Learning methods, which is why [6] provided an artificial intelligence platform around Bomberman including several associated intelligent agents and empirical experiments.

3 Approach

3.1 Reinforcement Learning Method and Regression Model

3.1.1 Features

This chapter describes how the game state is transformed into input features for the model. The encoding initially chosen is based on [15]. The dictionary containing the game state is transformed into one vector, containing the input features. The input feature vector consists of the following five independent matrices:

- Field state: free (0), breakable (1), obstructed cell (0.5)
- Player position: player (1), otherwise(0)
- Opponent positions: opponent (1), otherwise(0)
- Danger level of position: danger (1), no danger (0)
  Danger is caused by bombs on all fields an explosion can reach. Two aspects influence the value how dangerous a field is. The time until the bomb explodes and the distance from the bomb as escaping might

become more difficult the closer an agent is to a bomb. The following equation was derived for calculating the danger of a field for the field containing the bomb and the surrounding fields. The resulting danger score is normalized through the equation in the range between 0 and 1.

$$danger = \frac{\frac{time\_Passed}{time\_needed\_to\_explode}}{\sqrt{distance}}$$

Example 1: the bomb explodes after 4 time steps. Currently 2 time steps are over and the distance to the bomb is 3:

$$danger = \frac{\frac{2}{4}}{\sqrt{3}} = 0.28$$

Example 2: the bomb explodes after 4 time steps. Currently 3 time steps are over and the distance to the bomb is 1:

$$danger = \frac{\frac{3}{4}}{\sqrt{2}} = 0.75$$

- Desirability of position: coin (1), no coin (0)

These matrices are flattened and concatenated. This results in a feature vector containing 1445 elements (5*17*17).

Among other factors due to the high dimensionality the training process could be very slow. That is why the input state is further minimized. Franca, Paes and Clua [7] evaluated five different strategies for state representation (Binary Flag, Normalized Binary Flag, Hybrid, ICAART, ZeroOrOne). The performance of the different encodings is measured by the cumulative rewards during training in relation to the number of episodes. Their results were that Hybrid, ICAART and ZeroOrOne perform better than Binary Flag and Normalized Binary Flag. The best encoding regarding to [7] is ICAART. That is why the final implemented encoding is based on ICAART.

The matrices above encode three different types of information: general information (field state, player position, opponent positions), desired positions (Desirability of position) and dangerous positions (Danger level of position). The input state is minimized by concatenating general information and desired positions into one matrix and dangerous fields into another one. This keeps the balance between clearly separated information and maximal state minimization.

- General information and desired positions: free (0), breakable (1.5), obstructed cell (-1.5), own position (2), opponent positions (-2), coin (1)

- Danger level of position: danger (-1), no danger (0)

These matrices are flattened and concatenated. This results in a feature vector containing 578 elements (2*17*17).

As further optimization one could also include behavioral information as part of the features. E.g. one could encode the optimal path towards the coins as one feature or reactions after placing a bomb (run away) as another feature. Such a feature representation could be interpreted as step away from a machine learning based agent towards a rule-based agent as the task of learning the optimal movement was already encoded inside the feature space. In order to avoid this gray area of potentially violating the project guidelines it was decided to not implement feature representations, which enhance the state information by behavioral information.

3.1.2 Dueling Double Deep Q-Network

Based on the literature research, described in chapter 2, a dueling double DQN approach was selected. The concrete implementation and structure of this approach can be seen in figure 2. In the first layer, the feature vector (chapter 3.1.1) is inserted. The input layer is followed by a fully connected Dense layer. In the next step the dueling optimization is implemented. By the Lambda layer the input is split into a value and an advantage stream. The advantage stream consists of a Flatten layer, a Dense layer and a custom Lambda layer for reducing the mean. The value stream is first flattened by a Flatten layer and then fed into a fully connected Dense layer. Finally both streams are again merged for receiving the final Q values.

3.2 Training process

3.2.1 Exploration-Exploitation

When choosing the right method for the exploration-exploitation tradeoff, [15] gives an insight in how different exploration methods perform in the Bomberman environment (see Figure 3).

In the long run, Max Boltzmann performs best with

$$\Pi(s,a) = \frac{e^{Q(s,a)/T}}{\sum_i^{|A|} e^{Q(s,a^i)/T}} \tag{9}$$

and T being the temperature parameter. But as the result is based on 100 generations of training with each generation comprising 10.000 episodes, Diminishing ϵ-Greedy as the second best exploration method was chosen as this exploration method converges faster in the early stages of training.

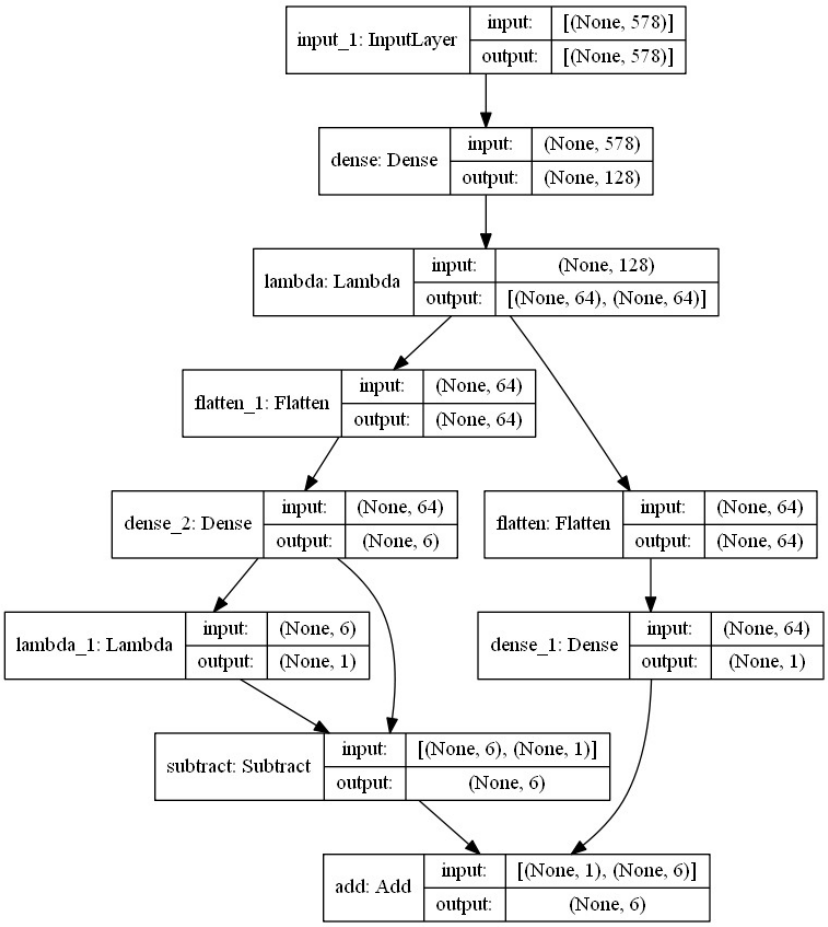


Figure 2: Structure of the DQN model

Two improvements where considered to optimize the exploration phase, but were discarded in the end. The first one is to replace the uniform sampling method by a multinomial sampling method in case an exploration step should be done, i.e. when a randomly generated number is smaller than ϵ. This means the second best action would be chosen more often compared to other actions during the exploration phase. This could be beneficial especially in later phases of training in case the Q-values are close to each other. But especially in the beginning of the training phase this could lead towards an unintended bias towards specific actions as the exploration of others will be suppressed probabilistically.

The second improvement was to include an exploration function as [8]

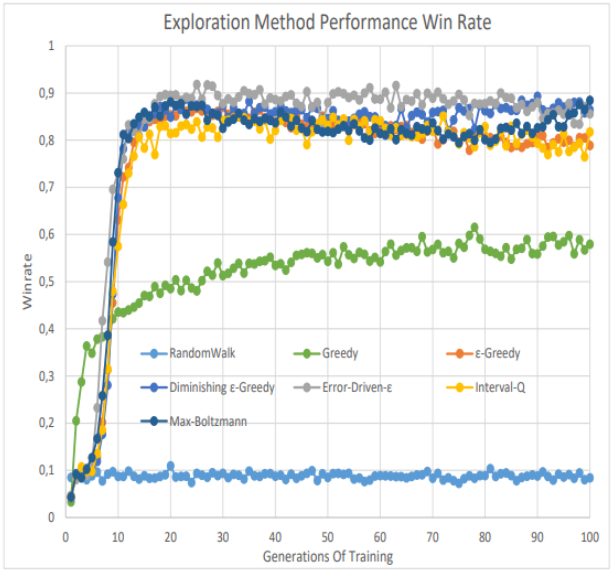


Figure 3: Comparison of different exploration methods [15]

proposes. A simple exploration function could be

$$f(q, n) = q + \frac{K}{1 + n} \tag{10}$$

with q being the Q-value and n being the count how often a specific action a was chosen in state s. K is a hyperparameter that determines the amount of curiosity during training. To implement this one would need to store n for every state and action. But as the state is far too high dimensional in the Bomberman environment this would require a lot of training just as using the Max Boltzmann exploration method to be beneficial in the end.

3.2.2 Prioritized Experience Replay Buffer and SumTree

Furthermore, a prioritized experience replay buffer was utilized to speed up the training process. To efficiently sample from it, a SumTree data structure was implemented inspired by [18], which is a binary tree whose parent nodes store the sum of its children. All leaf nodes of the SumTree store the priority of each temporal difference error which is the L1-norm between two succeeding Q-values. The SumTree inherently offers a stratified sampling method to sample experiences with a high temporal difference error and therefore high priority more often. Therefore the leaf nodes are grouped into sum segments with a sum value greater or equal a threshold value. Each segment can there-

fore contain a different amount of leaf nodes as priorities often differ in their magnitude. The amount of segments is determined by the demanded batch size and the threshold value by dividing the total sum of the tree (stored in the root node) by the batch size. From each segment one priority is sampled uniformly. As high priorities have less competitors in their segment, they will be sampled more frequently until they get overwritten.

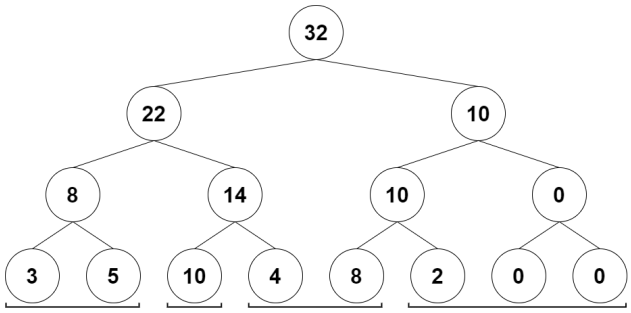


Figure 4: Stratified sampling of priorities from a SumTree

Figure 4 illustrates the process of sampling priorities with a batch size of four. One can see that the priority with magnitude ten will be sampled every time as it is the only priority within the sum segment. One can also see that in case the prioritized experience replay buffer is not filled, zeros might be sampled. To counteract this, the amount of sum segments to divide the SumTree into is the batch size plus one. Adding values to the SumTree has the complexity O(1) whereas subsequent updating the SumTree has the complexity O(log n).

The prioritized experience replay buffer only stores the priorities in the SumTree. The tuple (s, a, r, s') is stored in a separate list. To access the according experience tuple for a priority, one can easily calculate the acording index by $index_{list} = index_{tree} - size_{per} + 1$. Note that Equation 7 is used to calculate the priority value for each temporal difference error instead of Equation 5 as one would need to also sort the priorities in a different data structure which would add additional complexity and computing time. When sampling a batch from the prioritized experience replay buffer the tupel (s, a, r, s'), the according priorities, normalized weighting factors and update indices are returned.

Drawbacks of using a prioritized experience replay buffer over a normal experience replay buffer is the continuous maintenance of the SumTree data structure, which is currently updated every training step, i.e. every step in an episode. Updated value changes need to be propagated to the root node. This adds additional computing time but the time gained in training

progress by using prioritization should make up the time lost by maintaining the SumTree. Besides updating existing experiences every step also a new experience is added to the prioritized experience replay buffer every step which on the other hand executes fast.

3.2.3 Imitation Learning

For addressing the challenge of long training times, Imitation Learning can be used. With the rule-based agent, already a strong agent is present in the project. The idea of Imitation Learning is to speed up training by learning from existing behavior (e.g. from the rule-based agent).

The general applicability of Imitation Learning was evaluated by Hester et al. [12] and the application of Imitation Learning for the game Bomberman was examined by Franca, Paes and Clua [7].

Hester et al. [12] stated, that their Imitation Learning approach (Deep Q-learning from demonstrations) is better then a prioritized dueling double Deep Q-Network approach. Furthermore they have discovered, that the trained agent is able to outperform the behavior shown in training.

Franca, Paes and Clua [7] have found out in a setting comparable to the project that Imitation Learning followed by Proximal Policy Optimization (PPO) Learning achieves the best results. The PPO algorithm is a model-free Reinforcement Learning algorithm, which uses the advantage operator instead of Q-values.

As an optimization for the chosen approach (chapter 3.1.2), Imitation Learning based on [12] was implemented. This optimization is realized by splitting the training into a pretraining phase and a training phase. In the pretraining phase the Imitation Learning takes place. The experience replay buffer is filled by the actions of the rule-based agent and the DQN is pre-trained. In the training phase the standard Reinforcement Learning approach takes place. Based on the pretrained DQN and the filled experience replay buffer the training process is executed. For an evaluation of this training structure please refer to section 4.

3.2.4 Reward Shaping

Rewards are given when different kind of actions are taken or different kind of events occur. An overview of all rewards is given in Table 1.

Setting the rewards properly is a very difficult task and hard to evaluate without empirical study. Specifically for the Bomberman environment only few literatue was found that covers a reward function [15, 7]. All rewards are normalized to be in between -1 and 1 for more efficient training. Also focus

Table 1: Reward function

Reward	**Amount**
Movement in/out of danger zone	-0.3/+0.3
Movement towards/away from nearest coin	+0.1/-0.1
Movement towards/away from nearest opponent	+0.05/-0.05
WAITED	-0.2
INVALID_ACTION	-1
BOMB_DROPPED	0.4
CRATE_DESTROYED	0.7
COIN_COLLECTED	0.2
KILLED_OPPONENT	1
KILLED_SELF	-1
GOT_KILLED	-1

was set on giving reward feedback as soon as possible to guide the training process better towards the desired behavior.

In the first step the agent should be encouraged to gather coins, which is why the movement towards the nearest coin is rewarded. Nevertheless especially in the beginning of a game and also in late game phases setting bombs to find coins, to explore the arena and to kill opponents is even more crucial than finding coins. This is why the reward for dropping bombs is even higher than moving towards coins and gathering coins. To place bombs near creates the reward for destroying crates is set above the reward to simply dropping a bomb.

Having this setup of arranging rewards leads to a lot of suicide of the agent in the very beginning of the training phase even if the the own agents death is punished. Therefore rewards for moving out of danger zones and punishments for moving in danger zones are given to encourage the agent to walk away from bombs dropped. Note that this reward is even higher than the movement reward towards the nearest coin to keep an agent out of a danger zone in case a coin is located in the bomb radius.

Also an agent doing nothing or a invalid action should be punished by a negative reward. Therefore also no reward is given for rounds survived. Committing suicide or idleing was a big problem especially in the early stages of the training phase that should be countered using the introduced reward function.

Killing opponents gives the highest reward, which is exactly five times higher than collecting a coin just like the points given inside the game. Finally

to encourage an agent to also move towards an opponent agent an additional movement reward is given for moving towards the nearest opponents and a punishment in case the agent runs away from the nearest agent. Nevertheless such a reward is scaled quite low and only becomes more important in the late game steps. In the early stages of a game the agent should rather learn how to collect coins, how to destroy crates and how to survive.

One could also think of introducing a reward schedule that increases certain rewards during a game and lowers others. This could be applied when all coins are collected and an agents score can only be increased by killing opponents. Such a dynamic adjusting reward function still has to be evaluated.

3.2.5 Hyperparameters

Table 2 shows the choice of hyperparameters during training. As the computing resources are limited the hyperparameters were chosen to reach a fast convergence towards a first proper behavior which does not need to be optimal in the first place. [15] states to train an agent over 100 generations à 10.000 episodes whereas the win rate in Figure 3 is already quite high just after 10 generations. Having 400 steps per episode one generation can be trained locally within six hours which is really low inspite of using CUDA on a graphics card with a computing power score of 6.1 of 10. For further information one the local setup please refer to subsubsection 3.2.7. Note that the hyperparameters in Table 2 are the ones initially chosen, they were further adapted during the experimental phase in section 4.

The amount neurons per dense layer inside the dueling double DQN should not be higher than the input dimension itself as such a high degree of freedom is not needed to model the bomberman environment. For simplicity of the training process 128 was prefered over 256.

The Exponential Linear Unit (ELU) activation function was chosen to fight the dead ReLU problem and to speed up training by pushing the mean activation towards zero and therefore decreasing the bias shift from ReLU. To be more robust towards outliers a Huber loss function was prefered over a conventional Squared Error loss function. The learning rate was set quite high with 0.01 to reach a fast convergence towards a first playable behavior on limited computing resources.

The choice of the prioritized experience replay buffer hyperparameters was inspired by [18]. α was chosen slightly higher to once again speed up training through more prioritization. The β increment summand was chosen just as high to converge β to 1 at approximately the same time as ϵ converges to its lower limit after around 60.000 steps.

Table 2: Hyperparameters

Hyperparameter	**Value**
Dense layer dimension	128
Activation function	ELU
Loss function	Huber
Learning rate	0.01
α	0.65
β	0.4
β increment	0.00001
temporal difference ϵ bias	0.01
PER buffer min size	8.192
PER buffer max size	65.536
Batch size	64
γ	0.95
ϵ start	0.3
ϵ end	0.16
ϵ decay factor	0.99999

The size of the prioritized experience replay buffer needs to be a power of two to result in a balanced SumTree. The batch size should be lower than one percent of the total prioritized experience replay buffer size, which is why the minimum size of the prioritized experience replay buffer was chosen to be 8192. This leads to a start of the training process after around 21 episodes.

Choosing the right γ in approximative Q-learning is not too important as future rewards are approximated. Using a decaying factor of $\gamma = 0.95$ means rewards after 13 steps are only half that important as current rewards. ϵ is often decayed to around 0.05-0.2 and starts at either 1 or 0.3 [11, 15, 7]. As Imitation Learning in the pre-training phase already gives a good first behavioral model and also to not have too much random behavior in the beginning 0.3 was chosen as the start value for ϵ.

Furthermore, the model and the prioritized experience replay buffer are stored every 100 epsiodes in case the training process crashes because of unforeseen events.

3.2.6 Cloud Training

To further accelerate the training process and enable fast iterations for improving the approach in reasonable feedback loops, the possibility of training in the cloud was evaluated.

The prospects of success were shown by Lawrence et al. [16]. They evaluated the TensorFlow Deep Learning performance between CPUs, GPUs, local computers and the cloud. Their findings were that the usage of GPUs leads to a significant performance increase. Furthermore the cloud is not better by default. That is why it is recommended to compare the GPU performance metrics (e.g. compute capability score) of the local and the cloud configuration. The following table 3 compares common GPUs. Especially Nvidia is known in the area of GPUs for it's Compute Unified Device Architecture (CUDA). It provides a programming model and parallel computing platform for outsourcing processes to the GPU. In the area of Deep Learning there is the supplementary library called Nvidia CUDA Deep Neural Network Library (cuDNN), which provides hardware optimized routines for Deep Learning tasks.

	K80	P100	T4	V100	GTX 1080
CUDA Cores	2496	3584	2560	5120	2560
Tensor Cores	/	/	320	640	/
TeraFLOPS (Single Precision)	4,113	9,526	8,141	14,13	9,784
Memory Bandwidth (GB/sec)	240,6	732,2	320	897	345,6
Suggested Power Supply Unit	700	600	350	600	450
Compute capability	3.7	6.0	7.5	7.0	6.1

Table 3: GPU comparison
Based on [5] and [4]

Based on these basic information regarding training hardware and the associated findings, for the purposes of this project two requirements exist:

1. The Cloud GPUs should be much more performant than the local GPU (local GPU: GTX 1080). To notice visible improvements a compute capability of at least a score of seven is expected.

2. The GPUs could be used for free.

Hale [9] compares the following Cloud providers: Google Colab, Google Cloud Platform, AWS, Paperspace and vast.ai. In general either the first or the second requirement are not fulfilled by the different cloud providers.

This is exemplary illustrated by means of Google Colab and Google Cloud Platform. On the one hand Google Colab can be used after converting the project into one Jupyter Notebook. GPUs can be used for a limited time frame for free. The available GPUs are Nvidia K80s, T4s, P4s and P100s,

which are assigned automatically [1]. The compute capability score lies depending on the GPU between 3.7 and 7.5 [4]. So Google Colab matches the second requirement, but not necessarily the first one. Also much effort would have had to be invested into converting the project into a Jupyter Notebook. On the other hand the Google Cloud Platform, especially the AI Platform offers many performant GPUs to choose from, but it can quickly become very expensive, as the following example calculation shows [3].

The calculation was done with the Google Cloud Pricing Calculator [2] and following parameters:

- Region: United States
- Selected tier: BASIC_GPU
- Job run time in minutes: 4320min (3 days)
- => estimated costs: 71.93€

That is why the Google Cloud Platform matches the first requirement, but not the second one.

Based on the above findings the possibility of cloud training is not realized and the training process is executed locally.

3.2.7 Local Training Setup

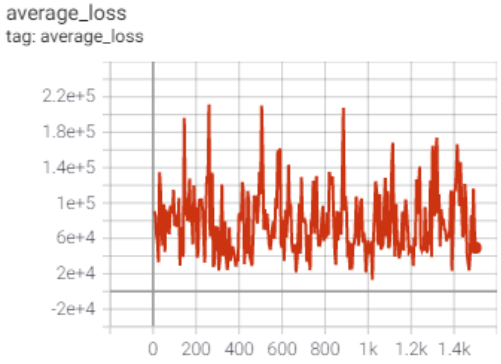
As mentioned in the previous chapter the local training setup is based on the GPU Nvidia GTX 1080 with a comuting power of 6.1 out of 10. Furthermore CUDA (version 11.2.1) and the cuDNN library (version 8.1.0) is used. In cooperation with TensorFlow (version 2.4.1) it is ensured that the local hardware is used optimally for the training process. The performance during training heavily depends on the progress a model has made so far. Pre-training usually takes longer as the full 400 steps are normally taken per episode which results in a performance of 1k episodes in nine hours. At the beginning of the main training phase, especially when the amount of exploration is still high and the model did not converge yet, 1k episodes can trained in less then one hour (approx. 35 minutes).

3.2.8 Training Visualization

To visualize the training process TensorBoard is utilized. Every five episodes the TensorBoard is updated with the average of the last five total rewards gathered over each episode, the average of the last five total episode losses, the current exploration factor ϵ and the current learning rate.

4 Experimental results

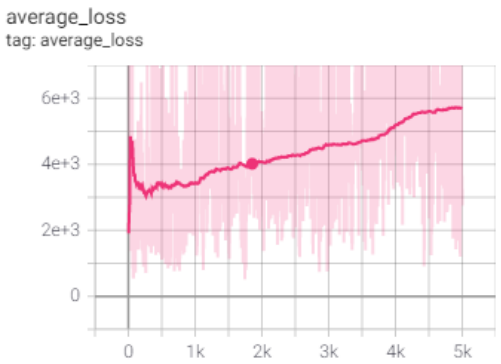
When starting the initial training process first a pre-training is executed which serves to fill the prioritized experience replay buffer with experiences from an expert rule-based agent and to pre-train the model itself to build up on some agent behavior already for the later training process. To not introduce a heavy bias the pre-training is stopped after 1.5k training games. Figure 5a shows the aggregated step loss per episode averaged after five episodes and Figure 5b the reward per episode averaged after five episodes.



(a) Pre-training loss

(b) Pre-training average reward

The two charts are not too surprising, the reward should on average be the same with a rule-based behavior. Also the loss does not decrease much with only 1.5k episodes of pre-training. When initiating the main training process with experiences gained from the dueling double DQN itself the following training curves can be experienced:



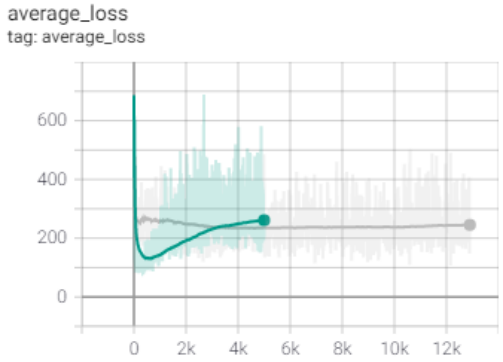
(a) Main-training loss

(b) Main-training average reward

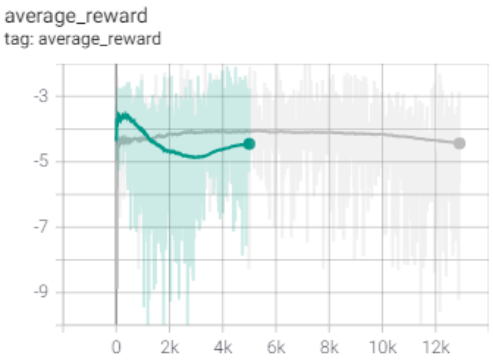
The complete opposite of what was expected was observed, the loss increased whereas the average reward decreased. One possible explanation

was a possible training bias introduced through the prioritized experience replay buffer towards transition with a high temporal difference error. Even if the gradient weighting factors were calculated and used to anneal the bias as proposed in the original paper, still this unintended behavior could be experienced. Therefore the training process was aborted after 5k episodes.

The next generation of the model was trained on a normal experience replay buffer with the following results:



(a) Main-training loss



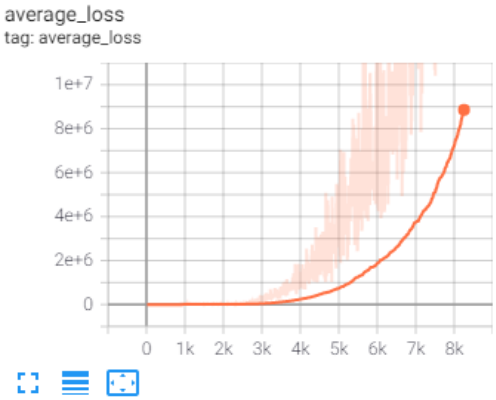
(b) Main-training average reward

Also, the initial exploration factor ϵ was increased from 0.3 to 1 to counter a possible pre-training bias and the maximum size of the experience replay buffer was decreased from 65.536 to 32.768 to train on more recent experiences. The green curve shows the training development until 5k episodes, which was then expanded to another 13k episodes plotted in the gray curve.

Having done these changes results in a far lower training loss curve which converges to an average episode loss of 250 per five episodes. Also the reward curve seems to converge at an average loss of about minus four. Nevertheless, this average reward still is not what was expected and when looking at the actual agent behavior, the agent always chooses to wait. A possible explanation for this might be that the agent tries to avoid future punishments by just waiting until it gets killed by another player.

To escape the bad local optimum and the small degree between rewards and punishments, the reward function was first slightly adjusted by increasing the magnitue of each reward by 100 and therefore breaking the normalization. This results in the two following training curves:

As one can see this negatively impacts the training behavior exponentially and got discarded right away. In the next try the adaption of the rewards were discarded again, the exploration factor ϵ remained at a start value of one and the maximum size of the experience replay buffer was left at 32.768. The amount of episodes to train during the main training was increased to



(a) Main-training loss

(b) Main-training average reward

30k. This equals three generations which should show at least some improvements during the training process as in [15], especially when increasing the exploration. The following training curves were experienced:

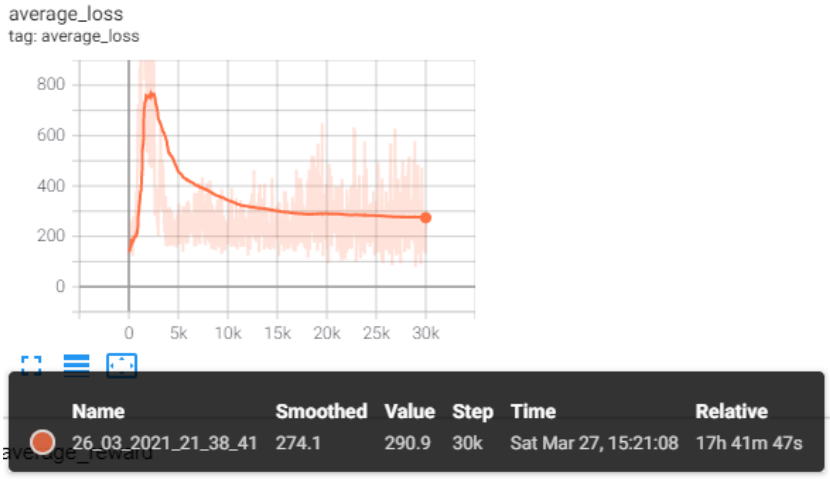
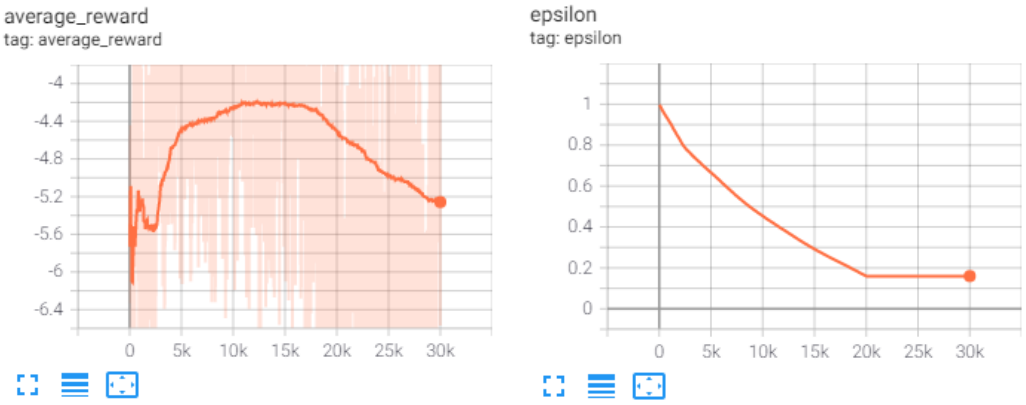


Figure 9: Main-training loss

When looking at the smoothed loss (see Figure 9) one can see the curve falls strictly monotonously from 1155 after around 2k episodes to 291 after 30k episodes. Having this setup taking the loss to around zero would still take ages when training locally. One could increase the learning rate, but previous experiments showed that this highly destabilized the training as the learning rate with 0.01 is already quite high compared to other papers [7, 15].



(a) Main-training average reward

(b) ϵ-decay

Having higher exploration did not fix the problem of ecaping the bad local optimum of the agent waiting until it gets killed. Instead it just seems to extend the training process as the loss is even higher than before. The course of the average reward curve (see Figure 10a) equals to the original one, but now after 18k episodes the average reward curve falls again. This development also seems to be independent from the exploration rate development (see Figure 10b).

To summarize neither changing the magnitude of the rewards, nor increasing the exploration rate helped to escape the bad local optimum. Only increasing the training episodes seems to very slowly decrease the loss which might result in another optimum in the future. But as such training takes a long time it was decided to not pursue further in the limited amout of time.

At this point the Bomberman scenario was simplified to only have one agent learning how to collect coins. Fortunately, this was achievable in the limited amount of time.

[Training curve + pictures]

5 Conclusion

Unquestionably, there is a lot of research provided in the field of Deep Q-learning achieving impressive results. Unfortunately, much less research is provided in common problems when facing Reinforcement Learning issues, instead the same problems are re-discovered over and over again. A. Irpan, software engineer at Google Brain, summarizes these problems to point research into a common direction [14]. Some of the problems described match with the ones met within this project.

One of them is the difficulty to design a reward function that matches

the given environment and domain exactly right. Otherwise shaped rewards might introduce a bias that overfits to certain actions. This was experienced as the agent overfits to waiting as to avoid future punishments.

Even when having a well designed reward function, still it might be difficult to escape a local optimum. This phenomenon originates in a bad exploration-exploitation trade-off with much exploration leading to misleading data and bad training results and much exploitation leading to burn-in behaviors. This could be tackled by using count-based or curiosity-driven exploration as introduced in Equation 10. Once again the behavior of the agent was also tried to be explained with this context. The bad local optimum was tried to escape by increasing the initial exploration rate ϵ while using the diminishing ϵ-greedy exploration method.

Even after a successful training one could not guarantee that the agent would have behaved the same way as during training. Especially when training against a certain other model these models could tend towards a co-evolution during training that makes them generalize weakly when being put into another environment.

To conclude the difficulties faced during training the agent using a dueling double DQN, one can conclude:

> *"[...] RL is very sensitive to both your initialization and to the dynamics of your training process, because your data is always collected online and the only supervision you get is a single scalar for reward. [...]. A policy that fails to discover good training examples in time will collapse towards learning nothing at all, as it becomes more confident that any deviation it tries will fail. [14]"*

Several improvements for future versions of the agent were collected. First the feature vector could be restricted to behavioral features instead of pure state features, which means pre-calculating certain tasks and only encode movement information and special game fields as features. This simplifies the learning process by not only minimizing the feature dimension but also by increasing the entropy of the features and therefore the degree of learning capability. Another way to alternate the feature vector is to use the actual encoded image of the playing field and giving it to a preceding convolutional neural network that is capable of recognizing certain patterns. Therefore similarities between states could be recognized and be encoded as the same feature vector that is then given to the normal dueling double DQN. This could once again minimize the feature space to important features only and therefore speed up the training process.

Next the reward function could be extended by dynamically adjusting certain rewards during the progress of a game like increasing the movement reward towards opponents. This has not become important so far as the agent does not reach the late game phase in the original environment.

This could be accomplished by utilizing count-based or curiosity-driven exploration to achieve a better exploration-exploitation trade-off or by pursueing more empirical experiments with different reward functions to escape the bad local optimum.

Last but not least, the episodes to train could be increased to evaluate whether the loss in Figure 9 would converge to zero and the reward in Figure 10a would actually start to increase again and therefore escape the bad local optimum of the agent always waiting.

References

[1] Google (editor). *Colaboratory – Google, FAQ*. URL: https://research.google.com/colaboratory/faq.html (visited on 03/24/2021).

[2] Google (editor). *Google Cloud Platform-Preisrechner*. de. URL: https://cloud.google.com/products/calculator?hl=de (visited on 03/24/2021).

[3] Google (editor). *GPUs zum Trainieren von Modellen in der Cloud verwenden*. de. URL: https://cloud.google.com/ai-platform/training/docs/using-gpus?hl=de (visited on 03/24/2021).

[4] Nvidia (editor). *CUDA GPUs*. en. June 2012. URL: https://developer.nvidia.com/cuda-gpus (visited on 03/24/2021).

[5] TechPowerUp (editor). *TechPowerUp*. en. URL: https://www.techpowerup.com/gpu-specs/ (visited on 03/26/2021).

[6] Manuel António da Cruz Lopes. "Bomberman as an Artificial Intelligence Platform". PhD thesis. Departamento de Ciência de Computadores: Universidade do Porto, 2016. URL: https://repositorio-aberto.up.pt/bitstream/10216/91011/2/176444.pdf.

[7] Ícaro Goulart Faria Motta França, Aline Paes, and Esteban Clua. "Learning How to Play Bomberman with Deep Reinforcement and Imitation Learning". In: *Entertainment Computing and Serious Games* (2019). DOI: 10.1007/978-3-030-34644-7_10.

[8] Aurélien Géron. *Praxiseinstieg Machine Learning mit Scikit-Learn und TensorFlow*. O'REILLEY, 2018. ISBN: 978-3-96006-061-8.

[9] Jeff Hale. *Best Deals in Deep Learning Cloud Providers*. en. Apr. 2019. URL: https://towardsdatascience.com/maximize-your-gpu-dollars-a9133f4e546a (visited on 03/24/2021).

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

[11] Matteo Hessel et al. "Rainbow: Combining Improvements in Deep Reinforcement Learning". In: *arXiv* (2017). URL: https://arxiv.org/abs/1710.02298.

[12] Todd Hester et al. "Deep q-learning from demonstrations". In: (2017). arXiv: 1704.03732 [cs.AI].

[13] Ying Huang, GuoLiang Wei, and YongXiong Wang. "V-D D3QN: the Variant of Double Deep Q-Learning Network with Dueling Architecture". In: 2018. DOI: 10.23919/ChiCC.2018.8483478.

[14] Alex Irpan. *Deep Reinforcement Learning Doesn't Work Yet*. June 2018. URL: https://www.alexirpan.com/2018/02/14/rl-hard.html (visited on 03/27/2021).

[15] Joseph Groot Kormelink, Madalina Drugan, and Marco Wiering. "Exploration Methods for Connectionist Q-Learning in Bomberman". In: 2018. DOI: 10.5220/0006556403550362.

[16] John Lawrence et al. "Comparing TensorFlow deep learning performance using CPUs, GPUs, local PCs and cloud". In: 2017.

[17] José Salvador, João Oliveira, and Maurício Breternitz. "Reinforcement learning: A literature review (September 2020)". In: (Oct. 2020). DOI: 10.13140/RG.2.2.30323.76327.

[18] Tom Schaul et al. "Prioritized Experience Replay". In: *arXiv* (2016). URL: https://arxiv.org/abs/1511.05952.

[19] David Silver et al. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* 362.6419 (2018). Publisher: American Association for the Advancement of Science tex.eprint: https://science.sciencemag.org/content/362/6419/1140.full.pdf, pp. 1140–1144. ISSN: 0036-8075. DOI: 10.1126/science.aar6404. URL: https://science.sciencemag.org/content/362/6419/1140.

[20] Ziyu Wang et al. "Dueling Network Architectures for Deep Reinforcement Learning". In: *arXiv* (2016). URL: https://arxiv.org/abs/1511.06581.