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.

Contents

List of Abbreviations

DQN	Deep-Q-Networks
MDP	Markov Decision Process

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. [7]

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. [9]

Salvador, Oliveira and Breternitz have summarized the evolution of Reinforcement in a literature review [7]. 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 we use Reinforcement Learning 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 kill all other players. Typically in a game round first the walls around the player are removed by placing bombs. Next the agent can navigates to his opponents kill them by placing bombs. [6]

After introducing the fundamentals and related work in chapter 2, our approach is described in chapter 3. Therefore our selected Reinforcement Learning method and the training process are presented. In chapter 4 the results of our 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 [3].

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

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 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 [11]. 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).

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

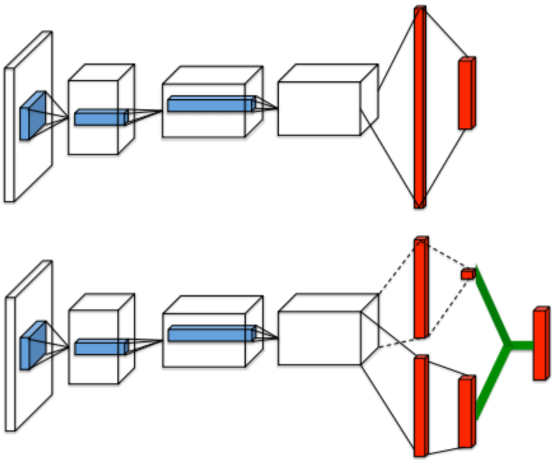

Figure 1: Architecture of a normal DQN (top) compared to a dueling DQN (bottom)

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 esspecially 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 [5].

Besides optimizing the learning process itself and the model architecture, [8] 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 esspecially 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 [4] 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 [6] 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, for each cell four values are computed:

- Free, breakable and obstructed cells are encoded as either 1, 0 or -1,
- The position of the player and opponent players are 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.

[2] 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 [1] 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. Our initial encoding is based on [6]. The dictionary containing the game state is transformed into one vector, containing the input features. Our input feature vector consists of the following five independent matrices:

- Field state: free (0), breakable (1), obstructed cell (-1)
- 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. We derived following equation for calculating the danger of a field for the field containing the bomb and the surrounding fields.

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

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

  $danger = \frac{\frac{2}{4}}{\sqrt{2}} = 0.70$

  TODO: normalized through equation TODO: advantage: gradation

- Desirability of position: desirable (1), not desirable (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.

TODO: adapt

- Field state: free (0), breakable (1), obstructed cell (-1)
- Player position: player (1), otherwise(0)
- Opponent positions: opponent (1), otherwise(0)
- Danger level of position: danger (1), no danger (0)
- Desirability of position: desirable (1), not desirable (0)

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

3.1.2 Double Dueling DQN

3.2 Training process

3.2.1 Exploration-Exploitation

When choosing the right method for the exploration-exploitation tradeoff, [6] gives an insight in how different exploration methods perform in the Bomberman environment (see Figure 2).

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.

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

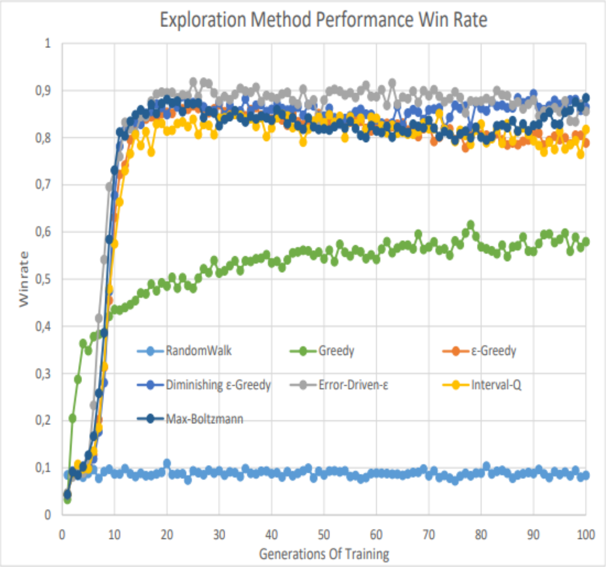

Figure 2: Comparison of different exploration methods

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 esspecially 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 [3] 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 [8], 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 therefore 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.

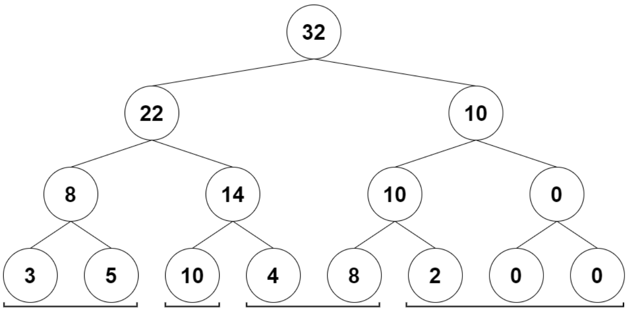

Figure 3: Stratified sampling of priorities from a SumTree

Figure 3 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(n) whereas 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. 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.

3.2.3 Imitation Learning

3.2.4 Rewards

3.2.5 Hyperparameters

3.2.6 Cloud Training and Training Visualization

4 Experimental results

5 Conclusion

A Appendix

A.1 Appendix A

References

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

[2] 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.

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

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

[5] Ying Huang, GuoLiang Wei, and YongXiong Wang. "V-D D3QN: The Variant of Double Deep Q-Learning Network with Dueling Architecture". In: 37th Chinese Control Conference (CCC). 2018. DOI: 10.23919/ChiCC.2018.8483478.

[6] Joseph Groot Kormelink, Madalina Drugan, and Marco Wiering. "Exploration Methods for Connectionist Q-Learning in Bomberman". In: 10th International Conference on Agents and Artificial Intelligence (ICAART). 2018. DOI: 10.5220/0006556403550362.

[7] 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.

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

[9] David Silver et al. "A General Reinforcement Learning Algorithm That Masters Chess, Shogi, and Go through Self-Play". In: *Science* 362.6419 (2018), pp. 1140–1144. ISSN: 0036-8075. DOI: 10.1126/science.aar6404. eprint: https://science.sciencemag.org/content/362/6419/1140.full.pdf. URL: https://science.sciencemag.org/content/362/6419/1140.

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