Fundamentals of Machine Learning
Winter term 2018/2019

UNIVERSITÄT
HEIDELBERG
ZUKUNFT
SEIT 1386

Final Project

Benjamin Bentner Carl von Randow Thomas Ackermann

Contents

1 Reinforcement Learning Method

1.1 Reinforcement Learning

In reinforcement learning our goal is to train a agent such that he finds the optimal policy $\pi(s)$. This means that the agent finds the best possible action a in each game state s.

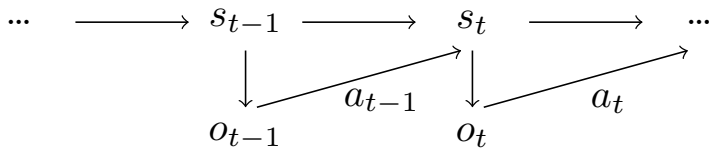

Figure 1.1: Diagram for reinforcement learning

1.2 Q-Learning

We used Q-Learning for our agent. The general Idea for Q-learning is to find the expected reward (or Q-values) for each action that the agent perform in a given game-state. If the space of states is small we could compute a matrix that would contain all Q-values. In each Game-state we would then take a look at the Matrix and pick the action that gives the highest expected reward.

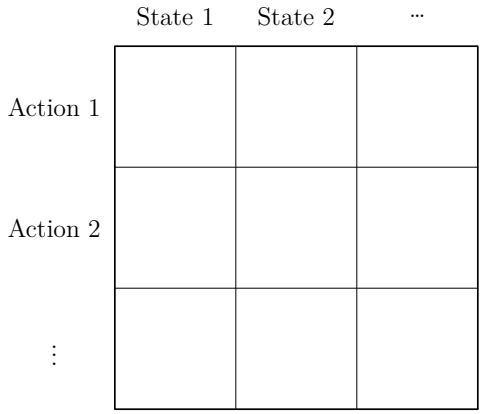

Figure 1.2: Q-table

The Q-Values can be computed in the following way:

$$\hat{Q}(s_t, a_t) = \hat{Q}_{\text{current}} + \alpha(y_t + \hat{Q}_{\text{current}}(s_t, a_t)) \tag{1.1}$$

In the first round the Q-Value will just be the reward that a specific action gave, but as the training continues the Q-values get more and more precise. The definition of y_t is different from algorithm to algorithm. At the beginning of our work we started with a 1-Step Q-learning algorithm. It's called 1-Step because we only use the expected reward of the next round. y_t is then defined as:

$$y_t = R_{t+1} + \gamma \max_{a'} \hat{Q}_{\text{current}}(s_{t+1}, a') \tag{1.2}$$

However, later on we moved to a n-step algorithm. As the name suggests, one computes the expected reward of the next n steps. In this case y_t is given as:

$$y_t = \sum_{t'=t+1}^{t+n} \gamma^{t'-t-1} R_t + \gamma^n \hat{Q}_{\text{current}}(s_{t+n}, a_{t+n}) \tag{1.3}$$

Unfortunately for us the number of states in Bomberman is far to big to use this strategy. What we do instead of calculating the entries of the Q-table is to define features which we use to characterize a game state. We define Q-functions that take these features as input and output our Q-function. So we shifted the problem from getting all Q-values to getting only some Q-values and fitting curves.

1.3 Regression Model

We need six different Q-functions for each of the possible actions (left, right, up, down, wait and bomb). In our case these Q-functions are linear functions, that take the fea-

tures x_i and the parameters $\theta_{i,a}$ and output our guess for the Q-value:

$$q_\theta(a, x_1, \ldots, x_n) = \theta_{0,a} + \theta_{1,a}x_1 + \cdots + \theta_{n,a}x_n \tag{1.4}$$

where n is the number of features. Because $x_1, \ldots, x_n$ describe our state s we will write $q_\theta(a, s)$ in the following.

In order to get the best possible parameters $\hat{\theta}_{i,a}$ we need to solve the following optimization problem:

$$\hat{theta} = \underset{\theta}{\text{argmin}}\ \mathcal{L}(q_\theta(a, s), Q^*(a, s)) \tag{1.5}$$

Here $Q^*(a, s)$ are the meassured Q-values and $\mathcal{L}(q_\theta(a, s), Q^*(a, s))$ is the Loss function which is defined as:

$$\mathcal{L} = \sum_t (\underbrace{\mu}_{weight} y_t - q_\theta(a_t, s_t))^2. \tag{1.6}$$

1.3.1 Levenberg Marquardt

We solved this optimization problem by using the Levenberg Marquardt Algorithm

2 Training Process

2.1 Features

2.1.1 Nearest Coin - distance

The first feature we implemented was the nearest coin feature. Since the first part of our training consisted of trying to collect all the coins in a game without opponents or crates and therefore without bombs the most crucial point we saw was the position of the coins on the gameboard. Since our primary goal at this point was to navigate to the next coin the most efficiently we ended up using only the position of the nearest coin regarding the agent position. Obviously we were aware that by taking only this information into account we would not always end up using the optimal way to collect all the coins but for a start we were satisfied with the use of this feature. In order to get the needed positions and differences we calculated the Euklidean distances from the agent position to all available coins and then chose the coin which had the minimum distance to our agent. In case of multiple coins with the same smallest distance we chose randomly(or in order?, depends on np.argmin). The Euklidean distance from the agent to the chosen nearest coin was our first feature and we thought about using ist in the reward section of the training for example by calculating the difference in the distance to the nearest coin after a step and rewarding or penalizing that step depending on wheater the distance grew or shrunk.

2.1.2 Nearest Coin - direction

In addition to the distance to the next coin we wanted to know, in which direction that coin lay. Therefore we calculated the difference in both the x- and the y-direction and divided the resulting coordinates by the total distance between coin and agent. That way we got the relative distance in each direction and could use that to navigate

better to the nearest coin. To achieve that we penalized the relative distance in both directions. For example we subtracted the relative distance in the x-direction from the reward for moving right and added it to the reward for moving left. Since the board position increases to the right and downwards and we subtracted the coin position from the agent position to get the direction this is the correct use of the distance in the rewards. The distance in the y-direction was obviously used equivalently. This evidently added a lot of information to the mere distance of the nearest coin because now the agent knew what way he had to go in order to decrease the first feature which he did not know before. This gave us features two and three.

2.1.3 Mean Coin

Already while implementing the first three features that would only take the nearest coin into account we realized that there could exist cases where those alone would not give us the shortest total way when collecting all coins. To make the agent better aware of this and for him to realize such situiations we wanted to create a feature that could inform the agent of the position of all the coins at a given time. Our aim was for him to realize multiple coins being positioned in a certain area and then check out that specific area to collect all the coins that lay there. In order to achieve that we summed up the difference between the agent position and the position of all remaining coins. We figured that it would make sense to give the closer coins a stronger influence than others which were further away. Therefore we divided the resulting distances in both directions by the total distance to the power of 1.5. The resulting vector would then to some extent point into the direction of the highest coin density. This vector we used similarly to the direction values for the nearest coin to help the agent sometimes move towards a higher coin density rather than the nearest coin.

2.1.4 Row-Column

After implementing the above features we started our first couple of training sessions to see how well they had effected the agent's understanding of the game. We noticed pretty quickly that the agent was trying to do the right thing however it did not realize wheather the way was blocked or not. For example if there was a wall between the agent and a coin it wanted to collect it would just keep running into the wall because according to it's features that would be the best way to get the coin. In an effort to solve this problem we thought of two more features that would indicate in which directions it would be bad to move depending on the row and column that the agent was located at that particular moment. To get those two features we analyzed the agents position. Looking at the rows for example the agent would not be able to move right or left in every second row. Equivalently it could not move up or down in every second column. Because of that we calculated the modulo 2 values of the agents position on the board.

The two features would therefore return the values zero or one depending on the row and column the agent was positioned in. We would then for example penalize the agent for trying to move to the right or the left while being in the second row. We hoped that this would reduce the number of invalid actions that our agent did significantly. One effect we did not take into account was the border of the game board. Say if the agent tried moving to the left while being in the top left corner of the board it would not get as big of a penalty then if he had tried doing so a row below. Since we penalized invalid actions anyways we decided to postpone adressing this problem and kept the feature with this flaw. At first we also wanted to award the agent for moving into possibly directions but then quickly discarded that idea because we would have actively rewarded him for running into the game borders by doing so.(Was that the reason?)

2.1.5 Results

2.2 Resulting θ_q

Our initial guess for our θ_q are shown in the following figure:

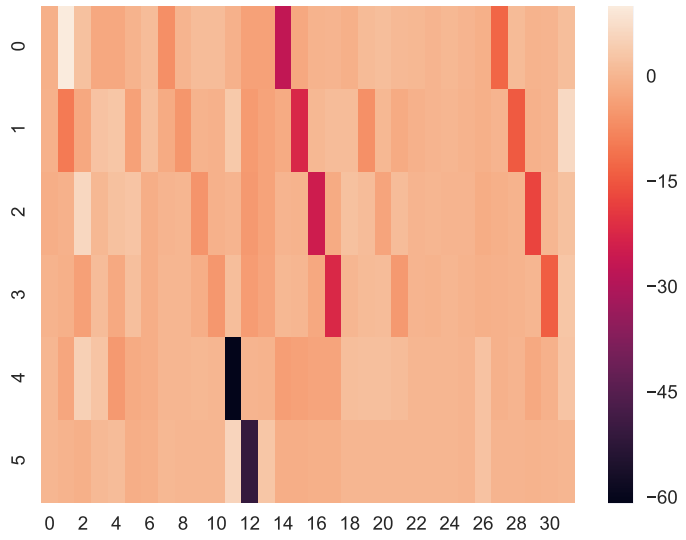

Figure 2.1: Initial guess for θ_q

After Training we got the following result

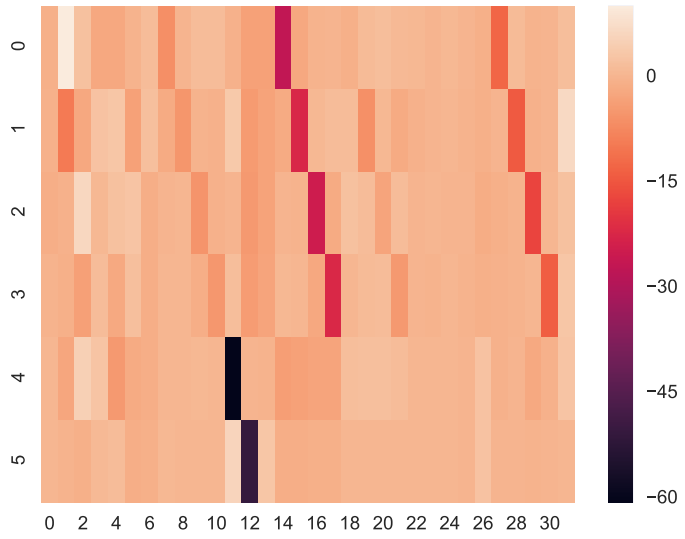

Figure 2.2: Initial guess for θ_q

3 Outlook

Bis jetzt immer theta genommen die durch alte feature entstanden sind. Komplett bei Null anfangen ist es dann besser oder schlechter?