Fundamentals of Machine Learning
Winter term 2018/2019

UNIVERSITÄT
HEIDELBERG
ZUKUNFT
SEIT 1386

Final Project

https://github.com/ThomasAckermann/ml_project

Benjamin Bentner Carl von Randow Thomas Ackermann

Contents

1 Reinforcement Learning Method (Thomas Ackermann)

1.1 Reinforcement Learning

In reinforcement learning our goal is to train an 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.

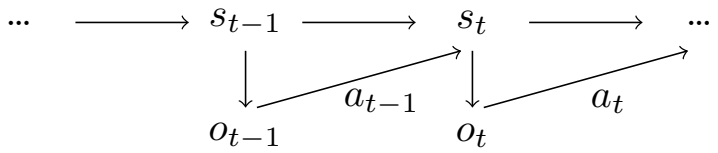

Figure 1.1: Diagram for reinforcement learning

The procedure can be seen in figure 1.1:
the agent starts in a state s_{t-1} and observes his environment o_{t-1} and gains rewards. With this information and the learned policy he chooses an action.

1.2 Q-Learning

1.2.1 Q-Table

We used Q-Learning for our agent. The general idea for Q-learning is to find the expected reward Q for each action that the agent can perform in a given state s.

If the space of states was small we could compute a matrix that would contain all Q-values (see 1.2). In every game state we would look at the Matrix and pick the action that gives the highest expected reward.

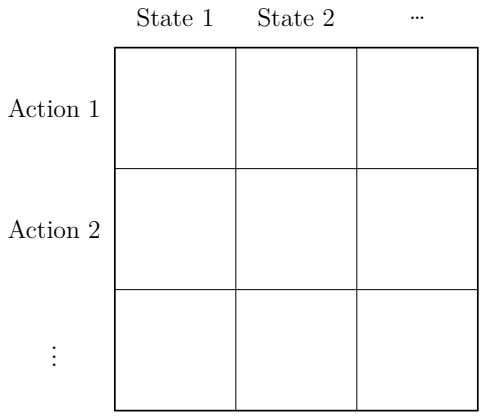

Figure 1.2: Q-table

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

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

The learning rate α is a hyper parameter and determines how fast the Q-values will change. 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. At this point, we can decide how we want to define y_t. We started with the following definition:

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

γ is a hyper parameter which acts as a weight factor for events that are further in the future. The Q-learning algorithm is called 1-Step Q-learning if y_t is defined in this way. The name comes from the fact that we only use the expected reward of the next round. 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:

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

We used $n = 7$ in the final version of our agent because in this case the Q-values will account for the end of a explosion process.

1.2.2 Q-Function

Until now we considered that we have a small number of possible states and can compute each Q-value. Unfortunately for us the number of states in Bomberman is far too big to use this strategy. What we do instead of calculating the entries of the Q-table is to define features and use them to compute the Q-values. Features are real values that are used to define a state. An example for a feature is the distance of the agent to the nearest coin. The features that we used for our agent are described in the next chapter.

If features are defined, the next step is to create Q-functions that take the features as input and calculate the Q-values from them. So we shifted the problem from getting Q-values for every possible state to getting only Q-values for states characterized by our features 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 features 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.
We used 33 features (which will be described in the next chapter) in total. We also tried other functions to model our Q-function. For example, we tried using a superposition of sigmoid functions but the results were worse than using a linear model (See Outlook).

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

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

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

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

μ is a weight factor that we can define.

1.3.1 Levenberg-Marquardt algorithm

We solved this optimization problem by using the Levenberg-Marquardt Algorithm. This algorithm solves the least-squares problem by calculating the Jacobian J of the modeling function.
The `scipy.optimize` module contains a method for solving least-squares using Levenberg-Marquardt (`curve_fit()`), which we used.

1.4 Rewards

In Q-learning our task is not only to find the right features to describe a state, but to also assign rewards for different actions. In each state we observe our environment (as described in figure 1.1), this also includes rewards.
There are many events that can occur where we could have given our agent rewards. However, we chose to use the following:

- moved left/right/up/down
- waited
- invalid action
- bomb dropped
- crate destroyed
- coin collected

These rewards are always assigned after the agent did an action.
For our agent, the values of these rewards depend on the values of the features in that situation. Feature independent rewards seem to have only very little effect on the learning of our agent. This is because in this case the agent can not understand how the reward is connected to the current game state and will therefore not learn how to improve.

1.5 Initial guess

To start the Q-learning algorithm an initial guess for the parameters $\theta_{a,i}$ is important because convergence will be much faster and more reliable. We trained our agent by increasingly adding elements to the game in the following way:

1. Agent and coins (probability for choosing 'WAIT' or 'BOMB' equal to zero)

2. Agent can wait and place bombs
3. Boxes are included
4. Enemys are inclued

We did this because we could always copy our parameters from the game modes where everything worked fine to the harder game modes.

1.6 Q-learning algorithm

Using all the information from the previous sections we can now describe the Q-learning algorithm, that we used.

Algorithm 1 Q-learning algorithm

```
Choose initial guess for θ
for episodes τ = 1, 2, ... do
    measure Q-values and features using exploration/exploitation
    calculate rewards
end for
use gathered data to update Q-values
solve optimization problem (eq. 1.4)
```

2 Features (Benjamin Bentner and Carl von Randow)

2.1 Nearest Coin - distance

(Benjamin Bentner from here) 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 game board. 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 euclidean 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.
The euclidean distance from the agent to the chosen nearest coin was our first feature and we thought about using it 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 whether the distance grew or shrunk.

2.2 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.

2.3 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 these features in and of itself 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 situations 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.
In the picture below you can see an agent trying to collect coins. The nearest coin would point him to the one directly above him and to the right. Mean coin would point him upwards and maybe slightly to the left but since the two other coins are further away they do not count as much as the nearer one.

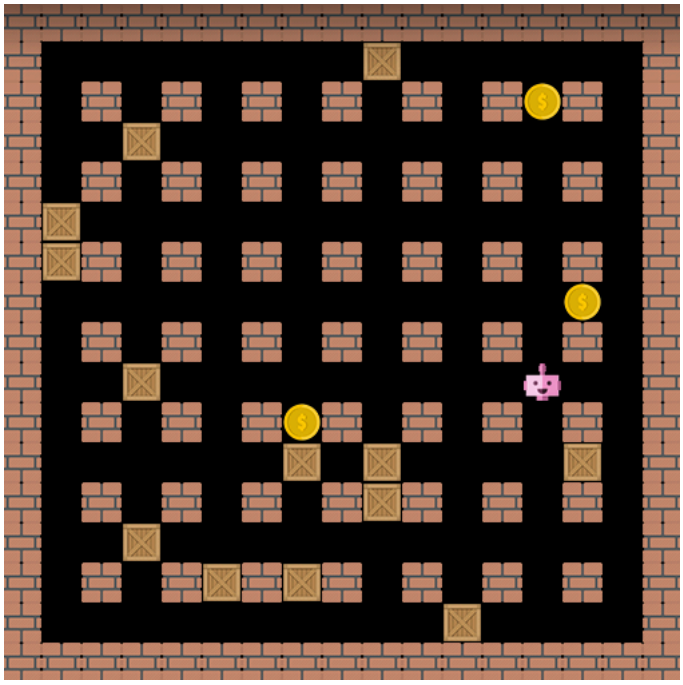

Figure 2.1: Coin positions

2.4 Row-Column

After implementing features from above, 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 whether the way was blocked or not. For example if there was a wall between the agent and a coin the agent 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 addressing 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.

2.5 Last move

At this point we thought that we were already pretty close to navigating the board effectively while collecting the coins. Training showed however that a lot of situations emerged where the agent just flickered back and forth between two positions and was only released from those positions when a random move occurred. We figured that this was partly the fault of the Row-Column feature because in a moment when the agent walked for example into a tile where the directions left and right were blocked it would get a reward for moving up just as much as it would for moving down. However the agent had to be coming from one of those directions and that could explain those flickering movements.
In an order to stop that, we changed the reward system to only penalizing it for trying to move into directions that would result in invalid actions. This removed the moving back and forth slightly but it could still be observed on multiple occasions. Therefore we implemented this new feature (or to be more specific four new features) that would indicate in which direction the agent had moved in the previous step. This was another feature that only took values one and zero and only one of the four features could be one in the same step. We designed the feature such that if the agent had moved left in the previous step the feature for moving right took the value one. We hoped that through penalizing moving right in the next step that the θ values for the features would be very negative for the opposing move (considering the move before) and approximately zero for the other possible moves. Looking at all four new features this would then give us the form of a four times four identity matrix with negative values on the main diagonal. After training the agent with the new features it wasted very little moves while collecting the coins on the board and we finally felt like we had found a way to do so efficiently. This was therefore the last feature we implemented for just collecting coins and the following features were used to destroy crates and evade the necessary bombs. While thinking about the crates and hidden coins underneath we already realized a potential problem with this feature. If the agent blew up a crate which contained a coin the agent would first have to hide from the explosion and the smartest thing to do after the explosion would be to run back to the position of the crate where the coin now lay. However to do so it would probably have to reverse

his last move at least if it wanted to go the shortest way. We hoped that this problem would be learned through training and that the reward for going as quickly as possible for the next coin would outweigh the penalty for reversing the last step. This was also a situation where we realized that n-step Q-learning would maybe help eradicate this problem.

2.6 Explosion Radius

Now the time had come to add 'WAIT' and 'BOMB' as possible action which we had so far ignored because we had had no use for them while just collecting coins. Since we wanted to start with very low crate density to slowly adjust the agent to the new situation our first intuition was that we would not really need the action 'WAIT' because the agent would almost always have a way of just running away from the explosion. However by adding 'BOMB' as an option we felt like there would be some major changes to the agent behavior. The first thing we felt was important was to know which positions would be in danger from a ticking or exploding bomb.
Therefore we created this feature that would return an array of all positions that could be reached by all currently active bombs in the game. We did this by looking three positions to the right, left, above and below the position of each bomb. As soon as we came across a wall we stopped going into that direction and this gave us the tiles that would be hit by the bombs explosion. In the picture below you can see two explosions that effect a different number of tiles due to their placement.

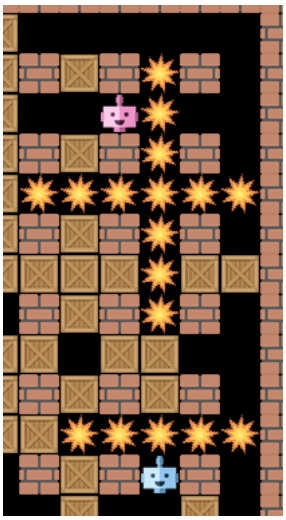

Figure 2.2: Explosion radius of two bombs

2.7 Position in Danger

The most important thing we realized we should add at this point was the fact that after placing a bomb some tiles would now be very dangerous to walk on even though they would maybe lead to the next coin. Therefore we added this feature that would indicate whether the current position of the agent or any position on the game board was dangerous due to a ticking or even exploding bomb (the feature would later take any given position as an argument and check the danger for that position which we used for other features that checked more than just the current position). In order to find out whether a position was in danger we compared the respective position to the array that was given by the explosion radius feature.
At first we thought this would be enough and we could just use values one and zero again but we soon realized that after placing a bomb the agent would necessarily move on dangerous tiles to escape the explosion. Therefore we gave the position danger several values depending on the state of the given bomb. We decided that we would take a certain value (for example 0.1) for a bomb that had a timer of four and then add to the danger depending on the timer. For example with every decrease in the timers number we would add 0.2 to the positions danger value. If the timer was down to zero or the explosion had the value two, meaning that it would still be deadly in the next step, the danger value would increase significantly and then go down to zero for an explosion with value one. This would not yet help us run away from the explosion since the agent could only realize how dangerous his current position was and not how to get to a safer tile but we hoped that it would at least stop the agent from waiting on dangerous tiles which he did in the beginning. We enforced this by punishing the action 'WAITED' severely with the amount of the danger on the current position. When we added opponents in the last phase we realized that we got problems if a position was threatened by more than one bomb. We eliminated those problems by always choosing the highest available danger value for one position.

2.8 Number of crates in Explosion Radius

So far the agent had just placed bombs through random moves because it had not yet learned how to place them strategically. This was something we wanted to change next because the agent would just make it more difficult for him to get the coins by placing bombs in positions that would destroy no or only one crate.
We thought that a good indicator on when to place bombs was the number of crates that could be destroyed by placing it on the current tile. Therefore we cross-referenced the array from the Explosion Radius feature with the state of the 'arena' matrix. If a position in explosion radius had the value corresponding to a crate in the 'arena' matrix we increased the number of crates by one and so we got the total number of

destructible crates. This would at least make sure that the agent would refrain from placing bombs without the possibility even destroying crates and maybe also stop him from placing a bomb every time it could destroy only one or two when there are better options like going for a coin that could otherwise be collected from an opponent if our agent wasted a step by placing a bomb. In an effort to enforce that we used the feature such that it would return the values 0.7 and 1.4 if one or two crates were destructible and 3 and 4 if three or four could be destroyed.
In the following picture you can see that the agent dropped a bomb on his current position. The value of our feature would in this case be five which makes the agent very inclined to placing a bomb.

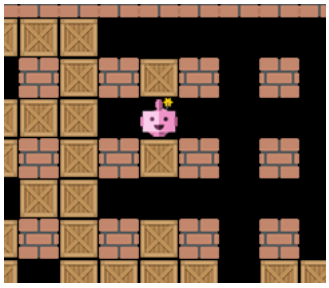

Figure 2.3: Number of destructible crates

2.9 Own bomb ticking

By adding the newest two features we soon realized that the agent tried placing bombs after he had already placed one that was at this point still ticking and through that it was wasting steps it could use to escape the explosion. This could however be changed in a pretty simple way. We again created a feature that would take values one and zero and simply check in the 'self' array of the game state whether it was possible to place a bomb at the current step or not. Our hope was that this would not effect any of the other possible actions and have a significant impact on the Q-value for dropping a bomb if one was already placed. To get this we punished the agent heavily for the action 'BOMB DROPPED' if the value of this feature was one.

2.10 Direction blocked

Also by adding even a small number of crates to the game board we realized that the Row-Column feature that we had implemented in the first phase of the game was now of little help since it completely ignored the crates. Because of that the agent kept running into crates and hence could not dodge the explosions or wasted a lot

of moves. We briefly considered removing the Row-Column feature completely but finally decided not to because at this point with the few crates it was still very helpful for exploring the game board and not running into all the walls again. We will talk about these two features in more detail in the outlook section. To be able to also detect tiles that were blocked by crates we created four more features, one for each direction. All these features would again take the values one or zero depending on whether the way was shut or not. We did not distinguish between crates or walls because for this feature it did not make a difference. Like for some previous features we hoped that the theta values would end up looking like a identity matrix with negative values since we wanted the feature to only effect the movement in the direction where it was blocked. In order to get that we penalized moving into a blocked direction. Later we also added opponents to this feature because it also makes no sense trying to run into an opponent. However, this could become a little tricky because the opponent could also move out of the way and make a seemingly blocked tile free again. In most situations treating the opponent as a wall makes sense though which is why we kept it like this for the final version.

2.11 Next move danger

Since we already had the feature position danger that gave us the amount of danger that the agent was currently standing on we thought it would be helpful to know what the amount of danger was on the surrounding tiles. So far our agent could not really know which way to go to evade a bomb since it only knew the danger of his current position which was something we figured we had to change. Once more we created four different features, one for each available direction. We wanted these features to indicate how high the danger level of the adjacent tiles would be in the next step.
Therefore we added plus and minus one in each direction to the current position and calculated the danger values with the Position danger feature using the altered positions. This would return us four numbers that hopefully would only effect the q-values of the according direction. For example we wanted to deduct from the Q-value of moving left if there was danger on the tile to our left but not change the other Q-values because of that danger. The amount of deduction depended on the level of danger that was returned from the Position danger feature. This worked very nicely and after training we could identify the negative unit matrix again. One could now wonder whether the agent would prefer to wait on a dangerous position to walking into another equally dangerous tile. We ensured that this was not the case by punishing waiting on a dangerous tile more severely than moving into one and by giving the action 'WAIT' a general penalty. This helped the agent massively in surviving his own bombs. In the following picture the blue agent is in a position where it has some danger from the bomb above (for example 0.1) and the tile below him is deadly for the next step. It

should therefore choose to wait in the next step. It also depicts a situation where all directions except 'DOWN' are blocked which would be given by the feature above.

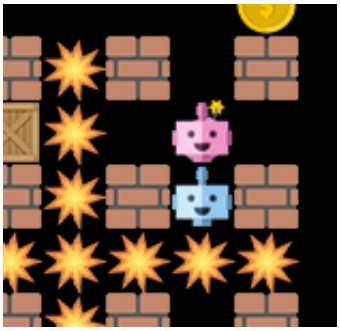

Figure 2.4: Dangerous positions and blocked directions

2.12 Crate difference

(Carl von Randwo from here)
While dealing with bombs relatively well now, the agent was still pretty lost when there were no coins available and no crates within the reach of its bombs. It seemed to just be content while sitting in a corner of the game board and was not seeking to destroy more crates to free coins. This was understandable since we had not yet implemented any features that helped him with that so that is what we did next.
We pretty much just copied the Nearest Coin and Mean Coin features from above and adapted them to help the agent move towards the remaining crates. Since we still wanted the agent to keep his focus on the coins as long as there were some to collect we did not reward these features as much which made the agent account the coins with a higher importance which was exactly what we wanted. Once there were no coins left, it started moving towards crates and tried finding more coins. One addition we made was that we let the agent know through another feature if the nearest coin was only one tile away. In that case we did not want the agent to further pursue closing the distance to the given crate but rather have it blown up. Therefore we gave it a high reward for dropping a bomb if this situation occurred. Considering all these features the Crate difference actually includes five separate features.

2.13 Dead end

At this point the agent was already behaving to our content when the crate density was low, however when we tried getting closer to the 70 percent that would apply in a real game the agent was too often killed by its own bombs. On the other hand we felt

like we had exhausted almost all possibilities for relatively simple features so we just looked at the situations where the agent died and tried to deduct from those which features we could still add. One reoccurring situation we observed was that the agent placed a bomb and then tried to escape to a part of the game board where he had no chance to survive the explosion.
This was due to the fact that he would run into dead ends where he had only up to three free tiles in on direction and could not go into the other direction. The picture below describes one of those situations which seemed to cause at least half of the agents deaths. Because of that we added four more features, one for each direction, that would indicate whether the road in that direction was a dead end or not. To get that information we started from the agents position and moved three tiles in each direction and checked the stat of that alley. For example if we wanted to check going up, we would first check the tile above and then the two adjacent tiles to the left and right. If the upper one was blocked we would immediately report a dead end. If not and the two tiles to the side were blocked we would continue one tile further up and repeat the process. If either the fourth straight tile in a given direction or one tile to the side were free we would return that it is safe to go into that direction. This gave us another identity matrix that prevented the agent from running into certain death which worked wonders for his survival rate.
In the picture below you can see that the direction right is a dead end. After placing the bomb the agent should not choose to move right.

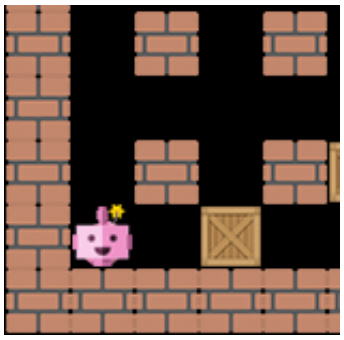

Figure 2.5: Dead end

2.14 No bomb

With the newest feature we often saw the agent survive through a good portion of the 400 game steps. However there were still many occasions where it was killed because of the dead ends. This happened almost always at the very beginning of the game. Because of that we only saw games that either ended within the first ten games or lasted through almost the entire game. Looking at the fatal game starts in more detail

we saw that the agent could create situation by placing a bomb where every available direction would become a dead end. To eradicate this mistake we created this feature that took the value one if the agent would create such a situation by placing a bomb. Therefore we used a heavy penalty for such a decision. Other actions should be almost independent of this feature.
The picture below depicts almost the only scenario in which this feature becomes relevant.

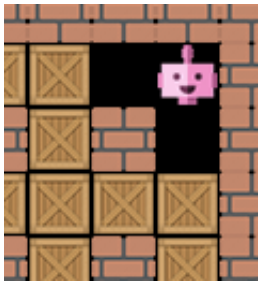

Figure 2.6: Possible creation of two dead ends

3 Training process (Thomas Ackermann)

3.1 Initial guess for θ

Having a good initial guess for the parameters θ can be really important as it can improve the speed at which we converge to our optimal parameters. We got our initial guesses by adding new features to the game as described before.
We started with no opponents, no crates, and no option for placing bombs.

3.2 Reward system

In order to train our agent we needed to assign rewards/penalties for good/bad behaviour. We used rewards/penalties that depend on the features because we want the agent to understand the connection between these two. In the following we want to explain what feature had an impact on which reward/penalty.

3.2.1 Movement (left, right, up, down)

Firstly, the Movement is influenced by the direction to the next coin and the direction of the mean coin, as well as by the direction to the next box. Coins were weighted twice as much as the boxes and mean values always much lower as the direction features.
Secondly, we checked if the agent is in an even or odd row/column number. The reward of this feature used to be really big but as our work continued the feature seemed to be not as effective as before and so we changed to have a rather small impact on the rewards.
Our feature for not letting him walk back and forth was also included in the rewards. However, the reward is not very high because in some cases it can be a good idea to do the opposite of the last step e.g. if an enemy places a bomb in the same direction as

the agent has been going.
The biggest penalties were given for next move danger and if the agent creates a dead end by placing a bomb. We did this because if one of these features is unequal to zero the agent will certainly die, so we want this to have a high influence on the agents decision making.
A small penalty was also given for each movement since we wanted the agent to find the shortest possibly way.

3.2.2 Waiting

The 'waited' event has only the position in danger feature as penalty, which is heavily weighted. There is only one case in which waiting in a dangerous position is a good idea and that is if the agent is in a position were the agent is in a area were a bomb will soon explodes but if he moves he will be in an explosion.

We generally penalized waiting since there are very few situations where it is a smart decision.

3.2.3 Invalid action

Invalid actions are always bad; that is why all features have a high penalty here. The feature with the highest penalty is the one for checking if the agent has an active bomb on the map. Because placing a bomb while another bomb is already ticking will always result in an invalid action and doing an invalid action while a bomb is on the map is extra dangerous because the agent needs to run away.

We also included the feature that checks if we are in an even or odd row/column with a low weight and the one that checks the blocked directions.

3.2.4 Bomb dropped

The most prominent features here are kill-able opponents and the feature for checking if the agent has a way to escape if he places a bomb now. These two features have the highest weight because we want our agent to live as long as possible.
The other two features indicate if there is an opponent that can be killed and if there are crates to destroy.

3.3 Data storage

In total we had three files were data was stored:

- `theta_q.npy`
- `q_data.npy`
- `all_data.npy`

The first file contains the current best parameters θ for the Q-function. The other two files contained the data of the games. Each step in the game, the following information was saved: chosen action, measured Q-values and all the associated features. `q_data.npy` contains only the last 6000 measurements and `all_data.npy` contains all data that was ever measured.
The data was always saved at the end of a game. During the game the data was stored in two attributes that we added to the `self` object (One attribute for the θ parameters and the other one for the other values). By doing this we saved time because we did not have to perform `np.save` function after every step.
We always used 12000 data points to create the fits for our Q-functions. Half of them came from `q_data.npy` and the other half was randomly picked from the `all_data.npy`. This procedure was done because on the one hand, we think that the latest data points are always the best because the measurements of the Q-values become better and better. However, we do not want the agent to 'forget' previously learned behaviour. The data from `all_data.npy` also helps us with stability. Because if the new Q-values all show a strange or undesired behaviour (because there was a problem with the fit for example) then there are still old values which are not corrupted.

3.4 Choosing α and γ

The learning rate α has the effect of making the changes in Q higher or lower. This means that if α is low we need more training. We set $\alpha = 0.04$, which is pretty low. The reason for our decision is that we think that keeping α low will keep the parameters θ from diverging.
The other parameter γ is responsible for weighting the importance of future events. In the end, we set this parameter to $\gamma = 0.4$. On the one hand, we did not want γ to be too high, because we think that the next step always provides the most information for what to do next. On the other hand, we saw a big improvement by adding n-step learning and did not want loose this effect.

3.5 Coping with diverging θ

During our training process we often had problems with diverging θ. We think this occurred because our least-squares solver found a wrong local minimum. We did a

few things to take preventive action. Firstly, we used new and old data. Secondly, we choose our training rate rather small. We also discovered that not doing the fit every round would also decrease the risk of diverging θ. In the end we did the fit only every hundredth round.

3.6 Choosing action for next step

If the agent always picks the action with the highest Q-value, it is unlikely that he will develop new behaviour. This can be fixed by adding randomness into the decision making of the agent. There are a few different things we implemented and tested. At the beginning we used the measured Q-values to construct a probability distribution for the action that he chooses. The task was to find a function that would map high Q-values to a high probability and low ones to a low probability. We decided to use a Sigmoid function because it provides these things.

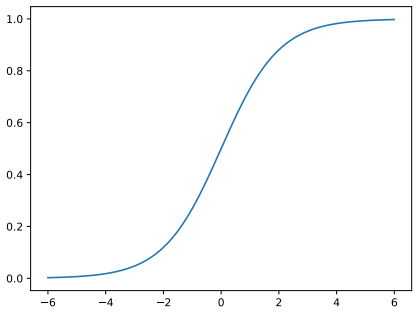

Figure 3.1: Sigmoid function

Later on, when we thought the agent already got a good idea of what works best we substituted this method by having a fixed probability for when to choose a random action and in the other cases it chooses the Q-value with the highest probability.

4 Results (Carl von Randow)

All in all, we are very happy with our results since our agent seemed to survive most rounds against the simple agents and could also play against itself pretty well. In the following we want to talk about the θ values that we got by training our agent and see how they make sense.

4.1 Resulting θ_q

After Training we got the following result which depicts our final θ values that we will use to play in the tournament.

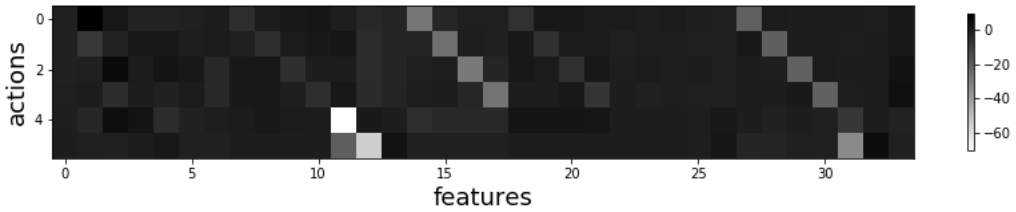

Figure 4.1: Final θ_q

In the following we want to take a closer look at why the final result looks as it looks. Let us start at the left hand side of the plot: We see the corresponding parameters of the coin direction features. The first column has no discernible structure, which we expect, because we did not use this feature to shape any rewards and the information could also be derived from the following two columns (i.e. there is no real ”new” information in this column). In the second column we can see, that if the coin is much more to the left or right than it is to the top or bottom (i.e. high value of the second feature,

low value of the third feature), we want to encourage the agent to go to the correct direction while simultaneously penalize the agent to go to the wrong direction. This means high values of theta, either positive or negative depending on the direction, for left and right, whereas the other actions should not be affected much by this feature. Correspondingly we want to do the same for the other direction, now for the ”up” and ”down” actions. The fourth and fifth feature should be roughly the same, but less prominent. One can not really see this in the plot though, because the expectation described above is only true if no crates (or other agents or bombs) block the way. Our θ depicts the case with crates etc., so it is understandable that the structure differs from our expectations.

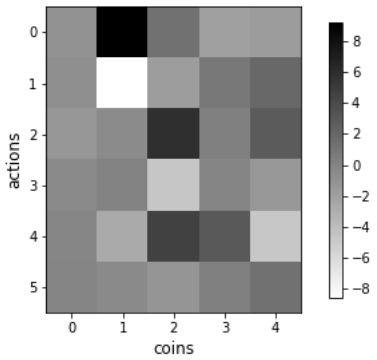

Figure 4.2: features 1 to 5

Next we see the Even/Odd parameters. In the case of no crates (which was the time we added this feature), we would expect the first two values of the first column and the third and fourth value of the second column to be the same value, because the corresponding actions are not blocked by walls (not taking into account the edge of the board). The other values of the first four rows should have the other sign because the actions are blocked. ”Bomb” and ”wait” should not be affected by these features.

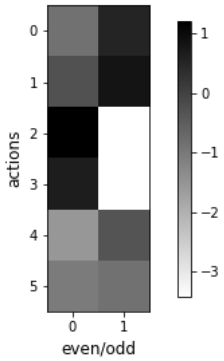

Figure 4.3: features 6 to 7

In the following figure one can see the four last move features. We expect a diagonal for the four moving actions because we penalized the agent if it directly reversed a move in the following move. All other entries should not have any high values, which is the case.

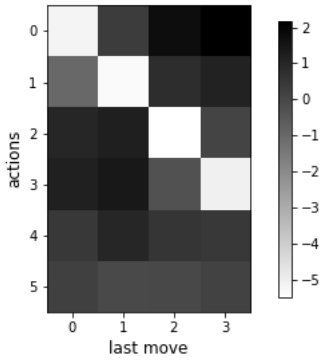

Figure 4.4: features 8 to 11

The ”wait” action has a high negative value if one is on an endangered tile, which can be seen very well. The ”bomb” action should be suppressed if an own bomb is ticking because this is an invalid action. The next feature depicts the number of crates destructible at the current position, which should urge the agent to drop a bomb. In other versions we were able to see this, but in this very plot the values of the other two features makes it practically impossible to glean anything from this column.

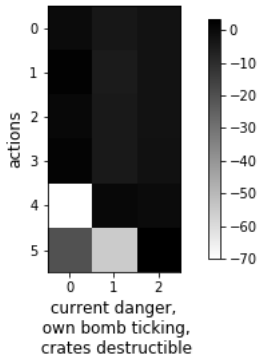

Figure 4.5: features 12 to 14

Next are the four ”directions blocked” features, which should again result in a diagonal, because a direction shall not be picked if it is blocked. We do not expect any real correlation between any other values in this figure.

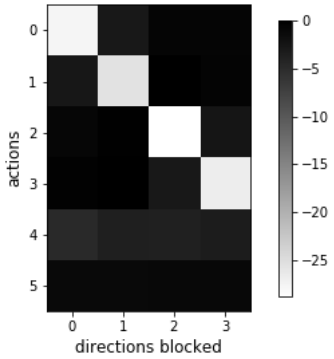

Figure 4.6: features 15 to 18

The ”next move danger” features are a diagonal as well, because an action is usually not good if there is danger in that direction. One can also see, that every direction that is endangered increases the probability to wait, which makes sense, because if all directions are endangered, ”wait” is the best choice.

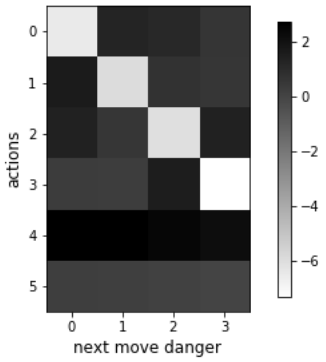

Figure 4.7: features 19 to 22

For the next four columns the expectation is identical to the coin features. In fact one can see it way better than in case of the coins, which is due to the fact, that the coins can be blocked by crates, which alters the best choice. The crates, however, can not really be blocked. If a crate is next to us (last column), the agent is encouraged to drop a bomb, in contrast to try to get even nearer to the crate, which is, of course, not possible.

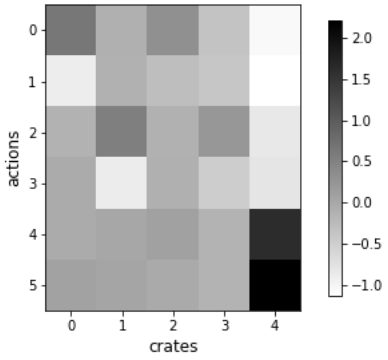

Figure 4.8: features 23 to 27

If there is a dead end in any direction after we dropped a bomb it would mean certain death in practically every case, which is why the diagonal is what we expect.

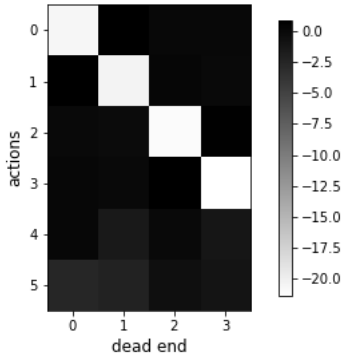

Figure 4.9: features 28 to 31

The ”no bomb” feature shall suppress the ”bomb” action, because dropping a bomb would leave us with dead ends only, which is to be prevented because it means certain death in most every case as well. The ”killable opponents” feature has not really been trained due to a shortage of time. We would expect no correlation to any action except for ”bomb”, but the latter should be favored. The offset has no real expectation, though ”wait” should be suppressed a little in general.

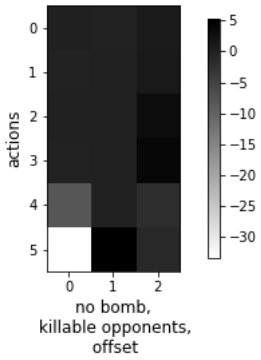

Figure 4.10: features 32 to 34

All θ values seem to match our expectations very well.

5 Outlook (all together)

In this part we want to discuss in what way we would have improved the agent more if we had had more time or what we would suggest for someone to look further into who wanted to use our agent and start working from its current state. We actually got a lot further with the project than we thought for a very long time. Although we started pretty early we did not really make a lot of progress until the last week before the deadline. Obviously at that point we spent a lot more time on the project than we had in the first couple of weeks. However it is still accurate to say that we made the most progress in the last quarter of the time we actually spent on the project. This was, of course, partly due to the fact that we had by then understood the theory better and that we had worked out a way to make changes pretty effectively. Another reason is, though, that the agent is now pretty stable and adding features does not cause him to change the whole parameter space. This was a problem we had almost until adding opponents. For those reasons I think it would very well be possible to impose further improvements on our agent that would not be too difficult to implement.

5.1 Training

One thing we could not do which we would have really liked was to train the agent against other AI opponents. The reasons we did not do that are first of all time problems because we only implemented the features for considering other players on the final day and the fact that we did not know any other groups who managed to finish the training.

5.2 Blocked vs. Row-Column

Like we already stated in the feature section we had a feature that told the agent to navigate between the walls of the board which we implemented while we were still only looking at coins. At a later stage where crates and even opponents were introduced into the game we created another feature that indicated whether a move in a certain direction was blocked by either a wall, a crate or an opponent. This feature obviously renders the first one meaningless since it covers all of the information given by the first one. In order to reduce our feature space which would improve calculation time and maybe make the fits more accurate in the remaining dimensions. We refrained from taking this step however because we were afraid of destroying some of the training that we had already made and therefore not being able to train other features as effectively since we would first have to retrain the new feature to cover for the part that the first one still had. Since it seemed more important to us to implement opponents and trying to kill them this is something we ignored.

5.3 Opponents as blocks

This is something we also mentioned in the feature section but we want to go further into it in this part. As the current state of the agent is concerned we treated other opponents equally to a crate or a wall in the corresponding feature. This is something we would maybe change in the future since an opponent can obviously move in contrast to crates and walls. For example if the opponent was oblivious to the fact that it could trap us by just remaining in his position or it is otherwise forced to leave its position an alley that was previously a dead end would then be available. This is something that could help our agent slightly because it would have a few more options for moving. However this would not always be the case which is why one would have to be very detailed when adding this.

5.4 Trapping opponents

The section of the game where we excelled the least in our opinion is trying to kill opponents. We would say that dodging opponents' bombs is something our agent does relatively well but it is not very good at aiming to kill opponents. All we added was that it should place a bomb if the opponent is currently in the kill radius of that bomb which does not take further movement of the other agent into account. This obviously makes it very easy for the opponent to dodge our bombs and it also limits our agents movement and can even lead to his death if other bombs are placed strategically. In the end all our agent does is place bombs that could influence the opponents somehow

but it might just as much hinder our own agent. Therefore we would propose that the agent could somehow learn how to place bombs with the sole purpose of killing opponents only if it will absolutely kill them. One option for that is using our dead end feature an checking if an opponent could get trapped by us dropping a bomb. This would then definitely be worth it because it would be a certain five points in the game which is almost enough to already win a round. If this was implemented we could tune the other bomb laying with the intention to kill down by a lot and actually focus on searching coins which should be more easy with less active bombs. However we feel like this is a stage where it would get really tricky with handwritten features and it seems to be that this would imply changes that could be better achieved by neural networks.

5.5 Relearn from zero

This is a very interesting aspect of the learning process which we have not yet understood completely. In the beginning we relearned everything when we added features which seemed to work fine. This was, however, while chasing coins and ignoring crates and opponents. As we got to the later stages and we thought it would slow the agent down if we let him relearn navigation on the board while also wanting him to learn about crates and bombs we decided to keep the already learned θ's for previously installed features and just add a certain amount of zeros or even initial guesses to the θ space. This resulted in the agent only changing those new thetas and not influencing the old ones too much. It would however be very interesting to see now that we have enough features to play the game relatively well how the agent would learn to act if we set all thetas to zero. We imagine it would take very long to find a stable fit and it must be much more difficult for the agent to distinguish reasons for rewards or penalties but it would be exciting to see whether it could at some point get to our current state or even exceed it because it would be able to find a better fit like this.

5.6 Long term/independent rewards

This is an aspect of Q-learning that we did not fully understand and that actually brought a few difficulties to our fitting. These troublesome rewards included in our opinion killing an opponent, getting interrupted, finding coins,surviving game and bomb exploded. First of all, we wanted to talk about getting interrupted. Is this really a penalty worthy event? We could not tell the agent to learn to calculate more quickly which is why we think it would have been confusing for it to penalize that. Killing an opponent and surviving a game was also a little confusing for us. It actually made sense to get a reward for killing an opponent but since that happens when we

place a bomb four or five steps earlier we think it might confuse the agent. In the step where we place the bomb and a few steps before it makes complete sense when you think about n-learning that it could feel the future reward of killing the opponent and would move into the correct spot where could kill someone. However in the actual step where we are rewarded the points it seems to be very confusing since the movement the agent makes while the bomb explodes will in most cases have nothing to do with the fact that the opponent just died and it might be rewarded for an action that was not actually that smart in the given situation just because somewhere else on the board an opponent ran into our explosion. The same goes for surviving a whole game. Again for the first couple of steps where this rewards starts to count it could make sense because the agent will choose actions that lead to him surviving the game but it would have massive consequences on the last say two steps that do not necessarily have anything to do with surviving the game and if the same features configuration were to occur in an earlier stage of the game another action would be smarter. The last two events we felt like were not meant for rewards were 'BOMB EXPLODED' and 'COIN FOUND'. Since the agent can only influence either placing a bomb or destroying a crate it has no power whatsoever over whether a coin was found or when the bomb explodes. Therefore we felt like it was enough that we rewarded/penalized the dropping of bombs and the attempt to destroy crates. Maybe some of these events were not meant to be rewarded or we did not understand them correctly but it still felt like we missed a few reward possibilities with these long term rewards. If we had increased the number of steps we regarded in the n-learning process they could have maybe been added in a logical way.

5.7 Fit functions

One area that we were eager to improve towards the beginning but then just accepted because it worked surprisingly well was the fit function we used. In the end we had a linear combination with all our features and an offset which gave us a 33 dimensional feature space and a 34 dimensional theta space. We think it is very plausible that this limits the accuracy of the fit severely and is definitely an are for improvement. At some point we tried using combinations of polynomial functions or sigmoid functions and thought about maybe also adding trigonometric functions but in the end we just used our linear combination because we had too little success in finding better functions. We still believe, though, that this can improve the convergence of the fit and therefore make the agents movement better.

5.8 Neural Networks

This is a very obvious improvement and one we do not feel the need to elaborate in more detail since we decided in the start against using them but it should be mentioned that neural networks could actually take care of a lot of these options. Like already stated more and better features can be found a lot easier by neural networks than by our imagination. Also neural networks do not necessarily use linear functions which would see our latter proposal for improvement done.

5.9 Evaluation of the project

Maybe this is a good transition to the evaluation and the best way to start is to give a short explanation why we chose against using neural networks. Our decision was not based on the naive believe one could obtain that it would have been more difficult, which may be the case but we do not know for sure. We actually thought about using neural networks for a couple of days and even started working on them as we were struggling with moving from phase one to phase two. In the end we made our decision in favor of Q-learning because the part that we thought was most exciting about this project was thinking of the features and finding clever ways of implementing and training them. Since neural networks would have done that without our ideas we had more of an actual impact on the agents intelligence like this.
Firstly, we want to thank the people a lot who created and thought of the way the project was implemented and presented. We noticed that a lot of thought had gone into it and pretty much everything seemed conveniently structured and easily accessible. This already saved us a lot of time because it did not take long to get used to the way everything worked. Also we feel like the whole idea of using this game (other equally competitive games would also be fine) as the project and especially making a tournament out of it was a very fun and refreshing idea. However, there are also a few points that we want to criticize and make suggestions for future lectures.
The first problem was the time frame and the scope of the project. The fact that it was placed between the end of most exams and two weeks within the start of the new semester seemed unfair to us since we felt forced to spend basically every free minute we had on working on the project which was made significantly worse by the pressure that was imposed by the idea of the tournament. We do not even want to imagine how this project was for groups of two people and for people who simultaneously had to go to work more than we did or study for later exams. Maybe we were just too slow but it really felt like the project was designed to make us work on it over the whole extend of the given time.
Our suggestion for future, similar projects which again we thought was very fun and would strongly suggest doing again, is that you could maybe start with the lectures

that were given in the end since you did not really need knowledge from the previous lectures to understand the project related ones, and then announce the tournament at the start of the semester. By reducing the workload during the semester a little bit and keeping the deadline that was given for the project you could make it a lot more manageable and students could actually plan the whole thing better instead of just being forced to sacrifice their whole semester break. In addition to that eight credit points would have seemed fair just for attending the lecture and the homework. Considering the amount of time we spent with the project it seems a little low compared to other lectures.

Just talking about the project itself we felt like we learned a lot and we would definitely be inclined to, given more time, work on it further to understand things even better. Using a game to do so and especially such a competitive one made working on the project significantly more enjoyable.