Final Project

Reinforcement Learning for Bomberman

Leonard Marks, Johannes Pfeil,

Maximilian Richter

`https://github.com/FranzWoyzeck/FML-Project-Submission`

Fundamentals of Machine Learning

Universität Heidelberg

Winter Semester 2020/2021

Contents

1 Introduction

by Maximilian Richter

The popular and well known classical computer game Bomberman is undoubtedly one of the best-suited environments to explore the realm of reinforcement learning and to test cutting edge machine learning models. Bomberman is certainly not unknown to the community and has already been studied in terms of reinforcement learning [2]. Not only does this game consist of a massive amount of possible discrete states, but also demands strategic behaviour in competition with multiple opponents. This is a difficult task to learn and to solve for a ordinary computer and there is no one-size-fits-all approach to this problem. However, recent developments in the on going research in reinforcement learning has shown promising results and brought up many remarkable strategies. Some of which we want to take a closer look at.

In this group project report we want to present the elaboration of our investigations to reinforcement learning methods as well as techniques to train a computer to play Bomberman in competition with AI agents, trained by other students. Chapter 2 is dedicated to our methodology and thus each section is authored individually in order to highlight the unique work and ideas of every group member. To still follow a common approach, the content of the feature and reward design for all our models was discussed collectively but formulated separately and can be found in chapter 3. In chapter 4 we take a look at our experimental results and the final choice of the model competing in the tournament. To conclude our work, the last chapter 5 is thought to give an outlook to further improve the performance of our agent and the project itself.

2 Reinforcement Learning and Regression Models

by Maximilian Richter

reinforcement learning stands for a variety of machine learning methods with the common objective to let an agent learn a strategy by itself. The desired behaviour of the agent is encouraged through rewards or penalties assigned to certain transitions or events. With this feedback, the agent is able to approximate the value of any given state or action and therefore maximize the reward over time (See Figure 1).

The framework of reinforcement learning is given in form of Markov decision processes. To discuss our presented work and results the following section is aimed to give a rough introduction to the notation of our report. For a more sufficient theoretical introduction to reinforcement learning and Markov decision processes we refer to more sophisticated discourses (e.g. [7]).

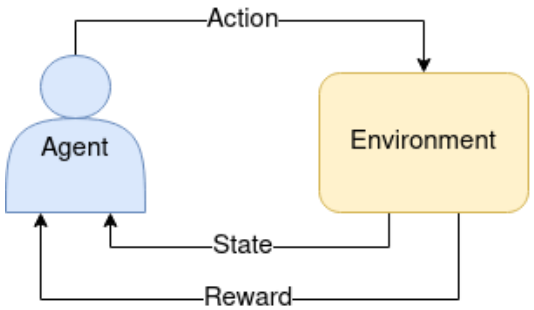

Figure 1: Flowchart of the reinforcement learning procedure

2.1 Markov Decision Process

by Maximilian Richter

A Markov decision process (MDP) is a 4-Tuple (S, A, T_a, r_a), consisting of

- S: State space
- A: Action space
- $T_a(s, s') = P(s' \mid s, a)$: Transition model, the probability of the transition (at time t) from state s to state s' under action a.
- $r_a(s, s')$: Immediate reward of transition s to s' after action a.

The solution to this decision problem is a function $\pi : S \to A$, called the policy, which maximizes the expected cumulative reward

$$\pi(s) = \arg\max_{a \in A} \mathbb{E}[R] = \arg\max_{a \in A} \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_a(s_t, s_{t+1})\right] \tag{1}$$

where the random variable R describes the return and is defined as the sum of rewards $r_t := r_a(s_t, s_{t+1})$ discounted by the factor $\gamma \in [0, 1)$

$$R = \sum_{t=0}^{\infty} \gamma^t r_t = r_0 + \gamma r_1 + \gamma^2 r_2 + \dots \ . \tag{2}$$

To solve the MDP we thus have to find an algorithm which maximizes this expected reward by following the policy π. A measure for the value of an given state is the state value function

$$V_\pi(s) = \mathbb{E}[R] \ . \tag{3}$$

However, in reinforcement learning the probabilities and rewards are unknown. In order to measure the value of a state given a specific action anyway, a further function is defined as

$$Q^* : S \times A \to \mathbb{R} \ . \tag{4}$$

At the beginning of a training this function is also unknown but can be learned by updating the Q-values with experience tuples (s_t, a_t, s_{t+1}, r_t), called Q-learning. The optimal policy is then given by

$$\pi^*(s) = \arg\max_a Q^*(s, a) \ . \tag{5}$$

reinforcement learning can hence solve Markov decision processes without general knowledge about the transition probabilities by training and exploring the space of possible states/action pairs.

In the wide area of machine learning several successful algorithms have emerged to train an optimal policy with reinforcement learning but unfortunately not enough time was available to be experimental with the choice of the learning model of our final agent. To address this problem we focused our common work on three popular approaches, namely Q-learning, Gradient Boost and Deep Q-learning. The following sections are intended to explore our different approaches with more detail.

2.2 Q-learning

by Leonard Marks

Q-values are the reward an agent can expect if it executes an action a_t and decides on an optimal decision.

- t: Step
- s_t: A state at step t
- a_t: An action at step t
- $Q(s_t, a_t)$: Q-value of state s_t with action a_t
- s_t: A state at step t from the set of States S

One of the easiest ways to directly learn the Q-values is Q-learning, with the function

$$Q^{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot \Big(r_t + \gamma \cdot \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \Big)$$

with γ as a discount factor, α the learning rate and r_t the reward received by moving from step t to $t+1$. Q-learning's weakness is that the agent has to learn $|S| \cdot |A|$ different parameters, for complex games which games states are almost impossible to reduce to important simple features, this method is not practical.

Q(s, a)

State (s) \ Action (a)	0	1	2	...
0	0.21772	0.09723	0.03119	0.04508
1	0.00168	0.01841	0.09021	0.03007
2	0.09021	0.03901	0.08233	0.06260
3	0.07745	0.06769	0.09672	0.09747
4	0.01272	0.05147	0.09482	0.08170
...	0.02254	0.02488	0.08215	0.03041

Figure 2: Q-table

Q-learning is a temporal difference algorithm, it is an agent learning from an environment through episodes with no prior knowledge of the environment, where the predictions are reevaluated after taking a step. If an episode is incomplete, it still generates an input for the temporal

difference algorithm. Hence it is not necessary to wait for the final reward. Using Q-learning, one can choose between two varieties: model-based and model-free. The model-free version lets the agent interact with the environment without using a pre-determined model. The interaction with the environment can further be split into off-policy (exploration can be unrelated to our image of an optimal action) and on-policy (agent decides on the best action by our decision). On the other hand if we determine the optimal course of the agent's action via pre-determined model it is called model-based.

Every time a game state happens, a state index is produced and the value of the reward of the following action from this state is then rewritten by the Q-learning function.

2.3 Linear Value Approximation

by Johannes Pfeil

With the help of linear value approximation, Q-values can be approximated. For each possible action, parameter vectors are trained. The Q-values for each action a and features $X(S)$ are then approximated as the dot product of the features and the parameter vector of the respective action (see equation 6). The agent can then decide for an action by choosing the action with the highest approximated Q-value.

$$Q(X(S), a) = X(S) \cdot \beta_a \tag{6}$$

with

- $Q(F, a)$: Q-value for states with features F and action a
- $X(S)$: Feature extraction function
- S: State
- β_a: Parameter vector for action a

The parameter vectors for each action are trained via gradient updates:

$$\beta_a \leftarrow \beta_a + \frac{\alpha}{N_{B_a}} \sum_{\tau,t \in B_a} X(S_{(\tau,t)})^T (Y_{\tau,t} - X(S_{\tau,t}) \cdot \beta_a) \tag{7}$$

with

- α: learning rate
- N_{B_a}: Number of batches for action a
- β_a: Parameter vector for action a

A disadvantage of this method is that the linear approximation is only successful if the features are well designed [3].

2.4 Gradient Boost

by Johannes Pfeil

Gradient Boost is an ensemble learning algorithm that outperforms random forests in accuracy. While in random forests all trees are independent of each other, in gradient boost the next tree is trained to correct the previous one. The algorithm requires a training set $\{(x_i, y_i)\}_{i=1}^n$ and a differentiable loss function $L(y, F(x))$ as input. The most common loss function for gradient boost is $\frac{1}{2}(y - F(x))$. To calculate the Q-values, we trained a gradient boost tree for each of the six possible actions, i.e. $y_0, ..., y_n$ are the Q-values for the respective action (see **??**).

First, the algorithm creates an inital model that estimates $y_0, ..., y_n$ by the same contant. Using the loss function $\frac{1}{2}(y - F(x))^2$, the estimated value is the mean value of $y_0, ..., y_n$.

$$F_0(x) = \arg\min_{\gamma} \sum L(y_i, \gamma) \tag{8}$$

Next, the pseudo-residuals are calculated. If $L(y, F(x)) = \frac{1}{2}(y - F(x)$, the pseudo-residuals are the actual residuals, i.e. $r_{i,m} = (y_i - F_{m-1}(x_i))$ where m is the m-th tree.

$$r_{i,m} = -\left[\frac{\delta L(y_i, F(x_i))}{\delta F(x_i)}\right]_{F(x)=F_{m-1}(x)} \tag{9}$$

A decision tree $h_m(x)$ is then fit to the pseudo-residuals. This new tree is then used to update the model according to the following formula

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x) \tag{10}$$

with

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)) \tag{11}$$

This results in the following pseudo code for creating a gradient boost regressor [1] (see **??**).

Algorithm 1 Gradient boost [1]

$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma)$ ▷ Initialize model

for m = 1 to M **do**:

$r_{i,m} = -\left[\frac{\delta L(y_i, F(x_i))}{\delta F(x_i)}\right]_{F(x)=F_{m-1}(x)}$ for i = 1,...,n ▷ Compute pseudo-residual

Fit decision tree $h_m(x)$ to pseudo-residuals

$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) + \gamma h_m(x_i))$

$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x)$ ▷ Update the model

end for

return $F_M(x)$

2.5 Deep Q-learning

by Maximilian Richter

Classical Q-learning suffers from memory limitations since every possible state/action pair has to be somehow represented in a table from which the Q-values can be extracted and updated. Deep Neural Networks (DNN) can be used in Q-learning to act as a function approximator mapping the given state/action pair to the desired Q-values. However, non-linerar function approximators are known to be unstable or even diverge if used without taking precautions. One of the most promising methods to stabilize Q-learning with DNNs is a combination of several techniques, described in the following, and patented by Google DeepMind as "Deep Q-Learning" [4].

In their paper Google addresses two key ingredients for stabilization: experience replay and adjusting the Q-values towards target values which are only updated occasionally. Both techniques are used to reduce correlations in the data distribution.

The first step for taking any action is to map the state s_t from the state space S to a n-dimensional

real vector, called the feature map ϕ

$$\phi : S \to \mathbb{R}^n, \quad s_t \mapsto \phi(s_t) = \phi_t. \tag{12}$$

The action of the agent is chosen by taking the maximum over the estimated Q-values

$$a_t = \arg\max_a Q(\phi(s_t), a; \theta) \tag{13}$$

where θ are the models trained weights. The feature map ϕ_t is then stored, together with its action a_t, its subsequent features ϕ_{t+1} and the collected rewards r_t from this transition in a memory M for the experience replay

$$\to \quad (\phi_t, a_t, \phi_{t+1}, r_t) \in M. \tag{14}$$

In order to perform gradient descent steps the mean squared error of the temporal difference error is evaluated

$$\mathcal{L} \sim \left(r_t + \gamma \max_a \hat{Q}(\phi_{t+1}, a; \theta^-) - Q(\phi_t, a_t; \theta) \right)^2 \tag{15}$$

differing from canonical Q-learning by $\hat{Q}$, the target network with weights θ^- from a previous iteration.

To further accelerate the training process, we use the Pytorch library for Python together with CUDA from Nvidia. As a starting point for the implementation we used the public Pytorch tutorial for Deep Q-learning [5]. As suggested in their tutorial we perform optimization by minimizing the Huber loss instead of the MSE and calculate the loss over a batch of transitions B, sampled from our experience replay memory:

$$\mathcal{L} = \frac{1}{|B|} \sum_{(s,a,s',r) \in B} \mathcal{L}(\delta) \quad \text{with } B \subset M \tag{16}$$

$$\text{where} \quad \mathcal{L}(\delta) = \begin{cases} \frac{1}{2}\delta^2 & \text{for } |\delta| \leq 1, \\ |\delta| - \frac{1}{2} & \text{otherwise} \end{cases} \tag{17}$$

Although Google achieves remarkable results with convolutional neural networks and multiple frames as input, we decided to follow our collective approach of carefully extracting features from the states by hand and to rely on a static feed forward network architecture. Our final network consists of $n = 21$ input neurons for each feature and $m = 6$ output neurons for each action as well as 2 fully connected hidden layers with 64 neurons each. The input values are either 0 or 1, so

Parameter	Value
learning rate	10^{-4}
minibatch size	128
replay memory size	10^6
target network update frequency	10000
discount factor	0.9
initial exploration	0.9
final exploration	0.01
exploration frame	$10^4 - 10^6$

Table 1: Parameters used for the Deep Q-learning approach.

a binary vector is evaluated as the state features and mapped to six real outputs, over which the maximum has to be taken. To find a suitable activation function several qualitative experiments have been carried out and the LeakyReLU activation has shown to yield the best results. As an optimizer the well-known ADAM optimizer with a learning rate of 10^{-4} is deployed. A full reference for all network parameters can be found in table 1.

3 Training Process

3.1 Feature Selection

3.1.1 Position Cases

by Johannes Pfeil, Leonard Marks

Considering the size of the playing field and the number of walls, there are 176 possibilities where the agent can be located. To reduce the number of states, we decided not to give the agent its absolute position and instead only give it information about its immediate surroundings. If we only consider the playing field with a box distance to the agent (horizontal, vertical and diagonal), we get a three times three boxes field. In total 15 different cases can occur, if we only regard the wall positions. Later we reduced this to four cases. Thereby we only considered whether the x or y coordinates of its position are even or odd. Because there is a wall on every position that has both an even x and y coordinate, there are only 3 cases left for the agent to be on.

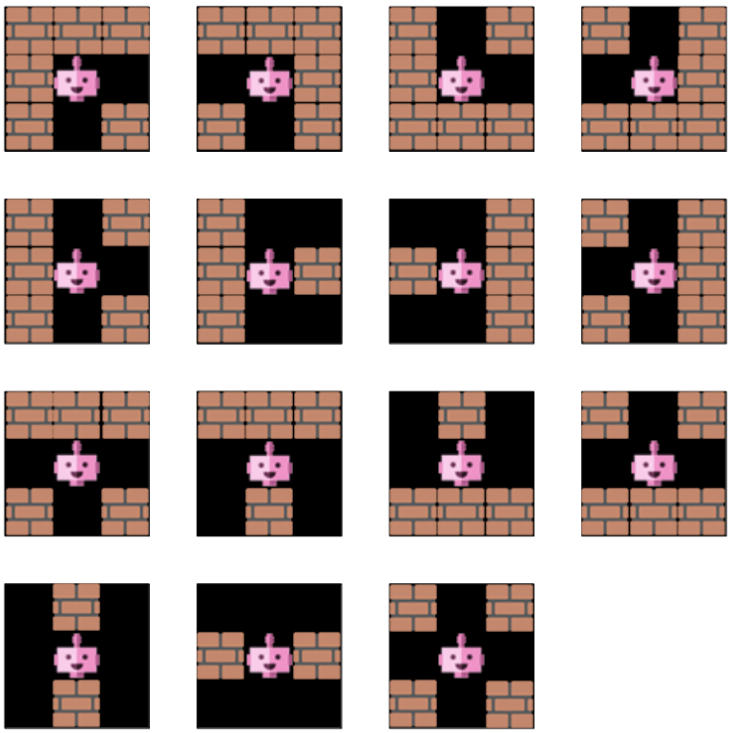

Figure 3: position cases

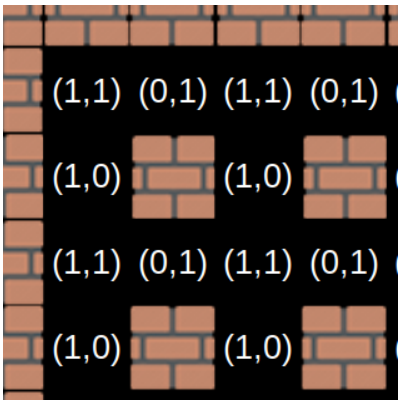

Figure 4: reduced position cases

As it is important to not only consider the agents global position, but also to use its relative surroundings, another method, which includes cases with random placed crates and transforms possible directions into an array of 4 entries, is more usable in some situations(figure 5). The array is sorted by the arrows starting from top and rotating counterclockwise. With this the agent has 16 unique states, which hold the information of direction possibilities, and thus actions which result in invalid ones are easily dropped in the learning process.

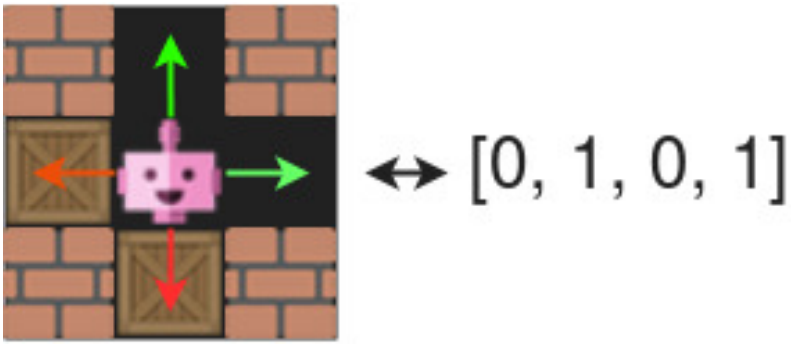

Figure 5: Direction Array

3.1.2 Direction to nearest Coin

by Maximilian Richter, Leonard Marks, Johannes Pfeil

If we would consider the whole playing field and every possible combination of nine coins, we would end up with

$$\binom{176}{9} = 362704129387600 \tag{18}$$

different possibilities to distribute those coins. In order to reduce this state space, we consider only the nearest accessible coin from the current position. Accessible means there exists a path without any crates. The relative distance vector of the position to the closest coin can then be used to compute the reduced features by looking at the sign of the components. For our two dimensional vector eight different cases can be distinguished. Those cases are represented by four feature dimensions.

Another approach we have used is not only to specify one of eight directions, but also to specify in which axis direction the agent must go furthest. This can avoid loops and detours, as the agent can now distinguish the case when a coin is two fields away in one axis direction and one field away in the other axis direction (see figure 6).

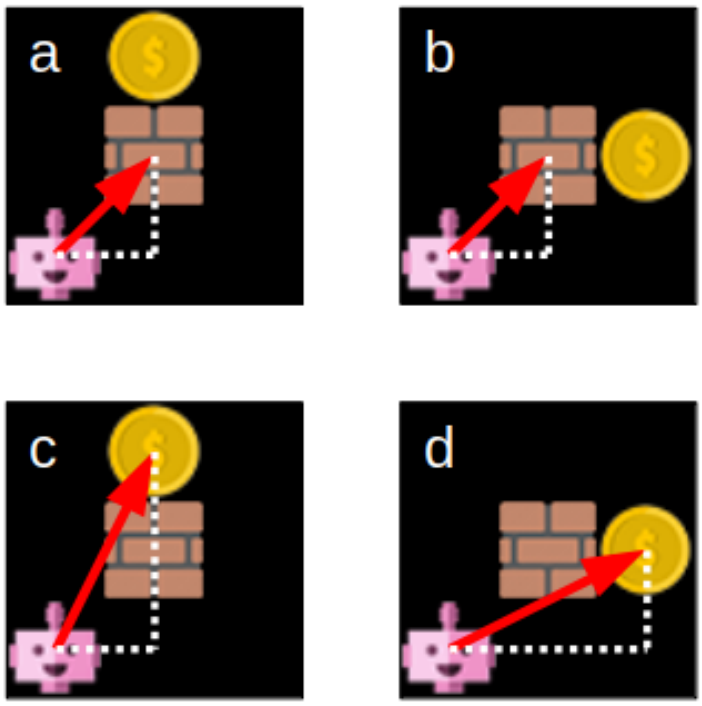

Figure 6: Critical positions for agent and coin. If the agent is only given one of 8 possible directions, it cannot distinguish between the cases in picture a and b. At this point, the agent cannot know whether it has to go up or to the right. If the agent is also given the information in which of the axis directions it has a further distance to the coin, the two cases can be distinguished (see c and d).

With this method there also rises the complication, that the agent looks at a coin to its right but

is not reachable by walking towards the right (figure 7). Together with the direction feature and the relative distance towards the coin the agent can learn to move on the green arrow. But this takes a lot of training time and can easily result in a loop if the rewards are not carefully chosen.

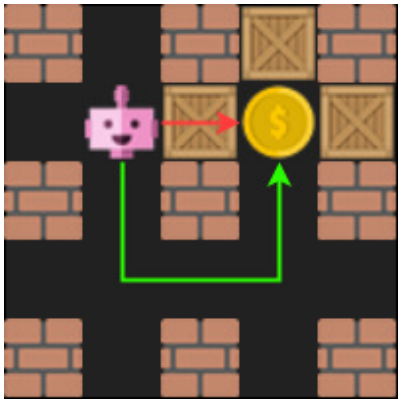

Figure 7: Relative direction not the direction the agent needs to follow to reach the coin

Furthermore if the coin is embeded in craits and walls it is not reachable and as such should not be considered by the agent until it is (figure 8)

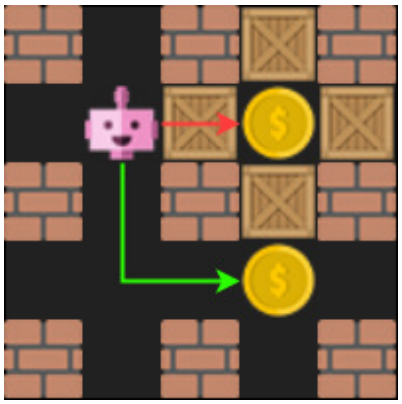

Figure 8: Agent fokuses on coin which is embeded by crates but doesn't consider reachable coin

3.1.3 Crate positions

by Leonard Marks

Directly at the start of a game one can see, that there are certain difficulties with the crates

position. One method is to calculate the manhattan distance of each crate relative to the player position and choose the closest relative crate in every step. This would cause loops, for example the agent would switch between two crates every step. A way around is to lock one specific crate even if another crate is closer after a few steps and only choose a new one when the size of all crates is changed. Furthermore rather than calculating the relative distance, the length of a path from a pathfinder algorithm is more sensible. All obstacles (crates, walls, etc.) are already put into consideration and if a crate is inaccessible the length of a path would be 0.

Figure 9: Locked Crate

In the left picture of figure 9 the agent locked the bottom crate as it is the closest, the top one is too far away and the middle crate is inaccessible. On the right the agent made a few steps but still has the bottom crate locked although the path (indicated by the green arrow) is longer than the distance to the top crate.
As seen in figure 10 directly at the start the agent has to learn to only drop a bomb if there is an escape route.
A simpler way is to think about the fact, that there is always a path where the player came from (more specific explanations in section 3.1.4).

With the relative direction to the locked crate helps to make the agent move towards the locked crate and drop a bomb if the agent is directly next to it. Now the agent only has to move out of the bomb's range as seen in figure 11

With this strategy the agent is fast to learn to bomb crates in an intelligent way, but still runs into problems as choosing a way away from the bomb which ends up in a dead end.

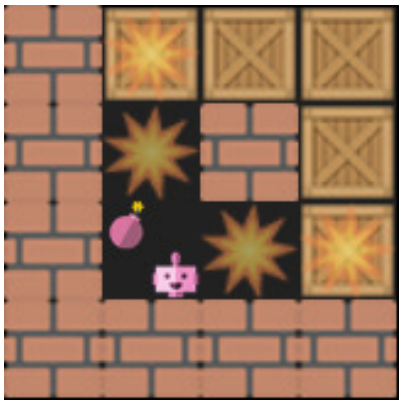

Figure 10: Start-position Issue

Figure 11: Ideal Start Situation

3.1.4 Danger zones

by Leonard Marks

The danger zone is defined by a list of tiles emerging from laid down bombs with the length of a bombs range. Every time a bomb is dropped it can be checked for its position cases, as there are only 3 cases this can be done in short time. The danger zone array is than enlarged by every tile the bomb can reach. If a new bomb is dropped without the other one exploding, the new bomb range is appended to the danger zone resulting in a list of coordinates which define the danger zone (figure 12, red polygon representing the danger zone list). Now it is easy to check if an agent is in danger or not Furthermore if the player is not in the danger zone the field is upgraded by new wall elements at position equal to the danger zone. Through this method the agent won't move into the zone as it would expect an invalid action, after sufficent training.

When the agent dropped the bomb himself, it is already in the danger zone and has to find a way to escape the zone. As seen in figure 13 there are quite a few situations and possible dead ends

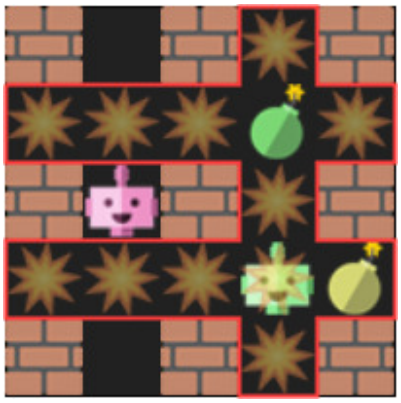

Figure 12: Danger zone polygon

(red arrow). The agent knows its 4 accessibility surroundings and thus only necessary to check for further free tiles. In case of figure 13 6 tiles are checked, 3 at each $x+2$ and $x-2$. If there is a crate at $(x+2, y)$ or $((x+2, y+1)$ and $(x+2, y-1))$ the tile at $(x+1, y)$ will become a wall, analogue for the other side. Now the agent thinks, likewise to the danger zone, that following the green arrow is the only possibility to not make an invalid action. This can be expanded for all 3 position cases of the bomb and further escape route situations.

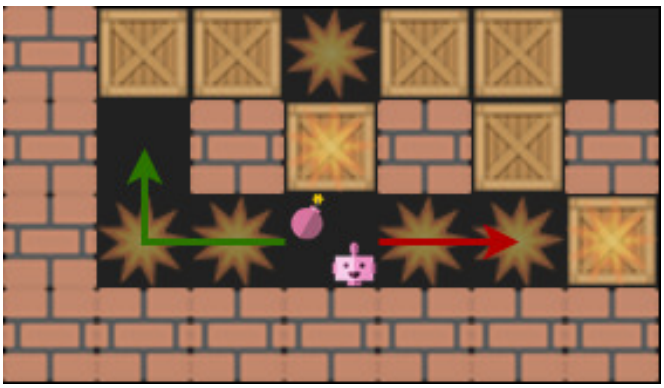

Figure 13: Only escape Route from danger zone

Another feature to boost the method is to give the agent its relative position to the bomb so that it does not decide to change his direction in the danger zone and focuses on reaching a diagonal relative position. This is a good feature for a single agent without enemies, as we only consider one bomb at a time, but as soon as enemies drop bombs close to our position there are a lot of cases the agent can get killed (figure 14). The only way for the agent to escape this

situation is by knowing where all the bombs are at every step, so that it doesn't move upwards towards the enemies bomb. If we take the direction array (figure 5) and expand it by 4 entries indicating the diagonals, going through all bombs the agent can set the values to 1 if a bomb is roughly in this direction. For the case in figure 14 the direction array is $[0, 0, 0, 1, 0, 0, 0, 1]$ as an example. Furthermore to reduce the complexity bombs, which are too far away, calculated via manhattan distance, to make an impact on the agent while their countdown, are ignored and not taken into account. So to summary if the agent drops the purple bomb and another bomb is top-right relative to its position and the agent is aware that it resides in a danger zone, it escapes by moving towards the bottom. There are still a few cases which are impossible to predict with this method (for example a crate is directly below the dropped bomb), these cases need more information regarding the playing field.

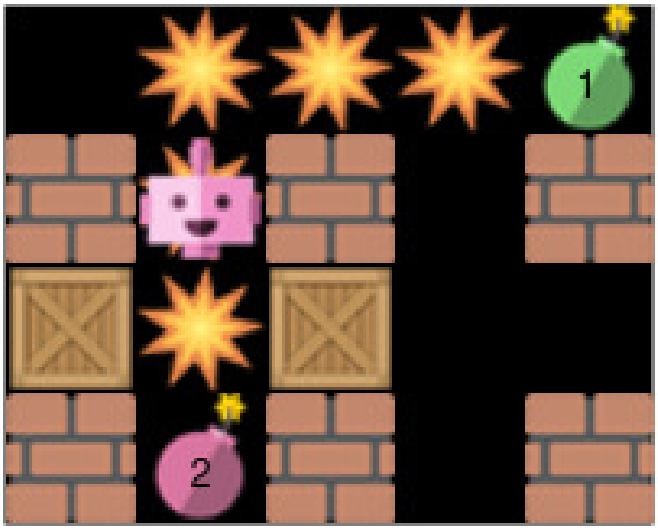

Figure 14: Definite death situation

3.1.5 Enemies

by Maximilian Richter

As soon as the agent has learned to efficiently navigate through the field and to destroy crates, it is a trivial task to extend the agent to fight against enemies. Our key idea is to store the enemies coordinates together with the crates and the enemies bombs together with the own bombs when evaluating the present state for taking the next action. This has the advantage that we can statically train the agent to run away from bombs and to walk towards and destroy targets.

Training with enemies can actually produce worse results, since the agent can not distinguish between walking towards or away from a target or if the target steps towards or away from the agent. The training with enemies could be realized by providing sparse rewards. However, in all of our experiments it has shown to be essential to assign auxiliary rewards. We have therefore decided to restrict the training of our final agent to collecting coins, destroying targets and running away from bombs.

3.2 Auxiliary rewards

by Johannes Pfeil

To speed up the learning behaviour of the agent, we have added additional auxiliary rewards. However, auxiliary rewards can also encourage behaviour that is not in line with the optimal policy. It is easier to set punishments because it is easier to find actions that cannot be part of a good policy than to find actions that correspond to the optimal policy.

3.2.1 Euclidean distance vs Manhattan distance

by Johannes Pfeil

When we speak of a distance between two points in the following, we mean the Manhattan distance

$$d(a, b) = \sum_i |a_i - b_i| \tag{19}$$

and not, for example, the Euclidean distance. We have chosen this because the agent can only move in a horizontal or vertical direction. The Manhattan distance between two points is therefore also the number of steps the agent has to take to get from one point to the other, provided it does not encounter any obstacles.

3.2.2 Shortest path distance

by Leonard Marks

As the manhatten distance does well in providing the distance to an object only by its x, y values, it does not take obstacles into consideration. Though we can add a +2 to the distance if a wall is between the object and agent, it doesn't tell us the real necessary path or if the object is not reachable.

The A^* search algorithm selects the path that minimizes

$$f(n) = g(n) + h(n) \tag{20}$$

where n is the next node on the path, $g(n)$ is the cost of the path from the start node to n, and $h(n)$ a self exploring function that estimates the cost of the cheapest path from n to the end. The algorithm provides a list of coordinates which indicate the path from a start point to the end point, `len(path)-2` the in-game distance and `len(path) == 0` if the object is reachable.

3.2.3 Moving towards coins or crates

by Leonard Marks

To drive the agent towards the coins, we rewarded the agent for each move that reduced the distance to the nearest coin. This reward has two disadvantages: First, it encourages greedy behaviour, so we cannot be sure that the agent is approaching optimal behaviour. Also, the agent could exploit the reward by moving in loops. We avoided the latter by punishing the agent when he moves away from the next coin. Same for the crates.

3.2.4 Moving away from bombs

by Leonard Marks

When the agent happens to be in a danger zone and only then, it must escape it which is only possible by moving away from the bomb. This is analogue to 3.2.3 but the other way around.

3.2.5 Danger-situations

by Leonard Marks

To force the agent to not (further) enter a Danger zone as well as multiplying the negative reward for unnecessary actions in the zone. The further Bomb reward must be set high enough to cancel out the danger reward so the agent is not punished for trying to exit the zone.

3.2.6 Waiting

by Leonard Marks

When the agent is only searching for coins, waiting is not optimal action, but as soon as an bomb

is placed, situations occur which make the waiting action necessary, especially close to danger zones. So only if no bomb is dropped, ticking or nearby Waiting is rewarded negative.

3.2.7 Dropping Bombs

by Leonard Marks

An agent who carelessly drops bombs in a great quantity is amusing to look at but not sensible to play the game. Thus a reward for good bomb drops is needed like placing a bomb directly next to a crate or with one free tile in between vertical, this is called a perfect move. As the agent should be encouraged to place bombs while exploring it is necessary to give it enough negative reward if it placed senseless and also important to give high positive reward if it is a good choice to place the bomb. By doing so the agent concentrates on moving towards a crate before deploying bombs, rather than dropping randomly and hoping to hit any crates. This is especially important for the start situation in figure 10 and 11.

3.2.8 Avoiding Loops

by Johannes Pfeil

When training the agents, we often encountered that the agent was doing unnecessary loops and wasting time without making any progress. To punish this, we could have given a penalty when the agent enters a field that he had already entered recently. However, this would have the consequence that we could severely restrict the agent if, for example, he wanted to avoid a newly appeared bomb. We therefore decided to punish the agent only if he re-enters a recently visited state. By using well-selected features, the agent will be punished if it arrives at a position again without having made any progress.

3.3 Exploration vs. Exploitation

by Maximilian Richter

In reinforcement learning we are faced with the problem of exploration versus exploitation. This trade-off involves the dilemma of whether to repeat a decision known to be successful (exploit) or to make an unexpected decision in hope of gaining more reward (explore).

There is a collection of well studied exploration techniques which are more or less successful,

Event	**Reward**
FURTHER CRATE	-10
CLOSER CRATE	10
CLOSER COIN	10
LOOP	-5
WAITED NO BOMB	-5
DANGER ZONE	-10
WRONG BOMB	-10
PERFECT MOVE	40
RUN AWAY	25
CLOSER BOMB	-15
INVALID ACTION	-3
BOMB DROPPED	-20
CRATE DESTROYED	5
COIN COLLECTED	10

Table 2: Summary of all events and rewards used in the final agent. The values presented should only be seen as representative and the used values can differ from model to model and from run to run. The given event/reward system has proven to work for our approach.

depending on the problem to solve and the complexity of the model. One of the simplest ways to achieve exploration would be to let the agent pick its action according to its Q-values with a probability of $(1-\epsilon)$ and choosing the next action stochastically with probability ϵ. However, this would lead to very long training times for new strategies, because for the agents initial Q-values it is practically impossible to have a meaningful policy which exploits anything. Therefore we tried other exploration techniques which are not static but evolve over the training in all of our approaches. We will discuss the most successful in the following.

3.3.1 ϵ-Greedy Exploration

The ϵ-greedy algorithm is probably the simplest evolving exploration technique. It works like the aforementioned random exploration, but decays the random probability ϵ over the training. The decay of ϵ is usually exponentially, although there are examples of linear or other decreasing functions.

The downsides of the ϵ-greedy method is similar to the static random exploration, such that it takes a long time to converge to a good solution. This is due the indiscriminative exploration, wasting resources by equally treating consequently worse options.

3.3.2 Max-Boltzmann Exploration

The Boltzmann exploration method solves the problem of the ϵ-greedy algorithm by assigning a probability to all actions. This method has shown to perform best in comparison with other strategies [2]. The probabilities are assigned using a Boltzmann distribution. The probability $\pi(s, a)$ for selecting action a in state s is:

$$\pi(s, a) = \frac{e^{Q(s,a)/T}}{\sum_i^{|A|} e^{Q(s,a_i)/T}} \tag{21}$$

with $|A|$, the amount of possible actions and the temperature T, an additional control parameter which can be evolved over the rounds. Starting with a high T and gradually decreasing it leads to an random exploration at first and a sparse distribution of actions later in the training. The improved exploration comes with a deceleration of the training process though.

This strategy can also be combined with the ϵ-greedy algorithm, such that we can, while decreasing T, also decrease the probability ϵ and therefore increasing the frequency of greedy decisions instead of the choosing according to the Boltzmann distribution.

4 Experimental Results

4.1 Coin Picking Agent

4.1.1 Q-learning

by Johannes Pfeil

In only 500 rounds we were able to train a Q-learning agent to reliably collect all the coins. After the training, the agent showed greedy behaviour by always going to the nearest coin and collecting it. The agent rarely made detours and did not make unnecessary loops.

As features, we only gave the agent the direction to the nearest coin, the axis direction in which the agent has to go the furthest (see chapter 3.1.2) and the own position case (see chapter 3.1.1). In order to train the agent to collect the coins as quickly as possible, we gave a slight penalty for each action the agent took, so that unnecessary detours are penalised. We penalised actions such as WAIT or BOMB particularly severely, as these actions are not useful for collecting the coins. We also penalised loops, i.e. the agent was penalised if it re-enters a recently visited state (see chapter 3.2.8). Because of the small number of features we had given the agent, there were only a small number of possible states. After 500 rounds, all states had already been visited multiple times and the respective Q-values were sufficiently approximated. Therefore, it was not necessary to apply a regression model.

4.1.2 Linear value approximation and gradient boost

by Johannes Pfeil

We used the simple Q-learning coin picking agent (see chapter 4.1.1) to test how well linear-value approximation works to approximate our previously used features. To do this, we had the previous Q-learning agent train the parameter vectors. After each action, we performed a gradient update of the parameter vector of the respective action. We did not let the agent use the new model to decide on an action, but let it continue to choose the action with the maximum observed Q-value. To measure how well the new model works, we compared the Q-values observed by Q-learning and the Q-values approximated by linear value approximation. The rate at which the agent would have made the same decision with linear value approximation converged to about 63% of the states. Because there are six possible actions, guessing would have resulted in a rate of

17%. Although our model is much better than guessing, it's still not reliable. Instead of adapting our features so that they can be better approximated, we decided not to pursue this model any further.

Similarly, we tested whether gradient boost is suitable as a regression model. Again we used the simple Q-learning agent (see chapter 4.1.1) to train the regressor. The rate at which the agent would have made the same decision with regression through gradient boost converged to about 82% of the states.

4.1.3 Deep Q-Learning

by Maximilian Richter

Discussed above, establishing a reward system which lets the agents learn the right policy to collect coins as fast as possible is a difficult task. So, as already mentioned beforehand, we decided to target one specific coin at a time to reduce the state space drastically. Additionally the agent only sees one of eight directions in which the closest coin lays. The agent then has to be rewarded by taking steps in the right direction, while not restricting the choice of the path. Constraining the rewarded path on the optimal path can, due to considering only the directional information to the coin and its absolute position, be ambiguous and should therefore be avoided. The agent quickly learns to aim towards its closest coin but struggles to realize that sometimes there are walls in between the agent and the coin. This has shown to not depend on penalizing the loops and could only be circumvented by training the agent for approximately four times as long as learning to walk towards coins. For an ϵ-greedy algorithm this took around 10^5 to 10^6 rounds in the case of Deep Q-learning. Resulting plots for the average reward per round and the average Q-values outputted by the network are displayed in Figure 15 and 16.

To further accelerate the training process it has shown to be helpful to lower the maximum step size to 100 steps per round, since the minimum steps to reach the goal of collecting all coins ranges from 40 to 60 steps and all subsequent steps are redundant or even counter productive for the learning process.

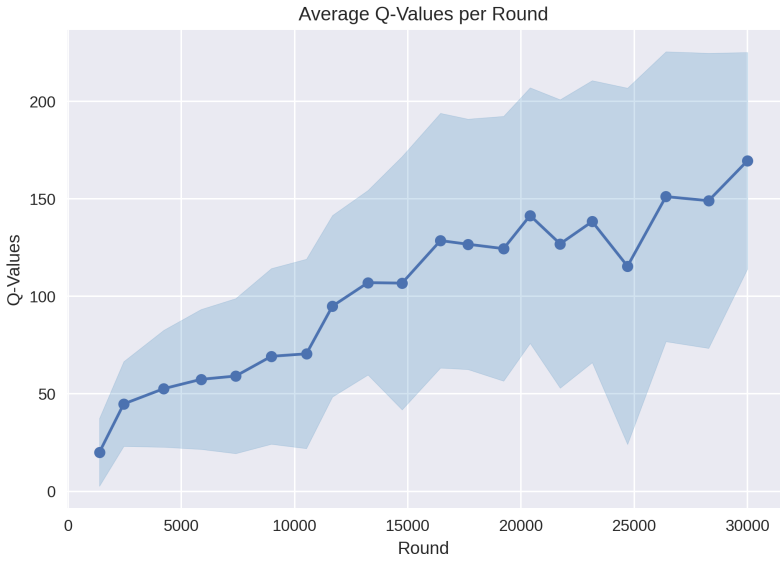

Figure 15: Average Q-values over rounds for coin collection with the ϵ-greedy algorithm

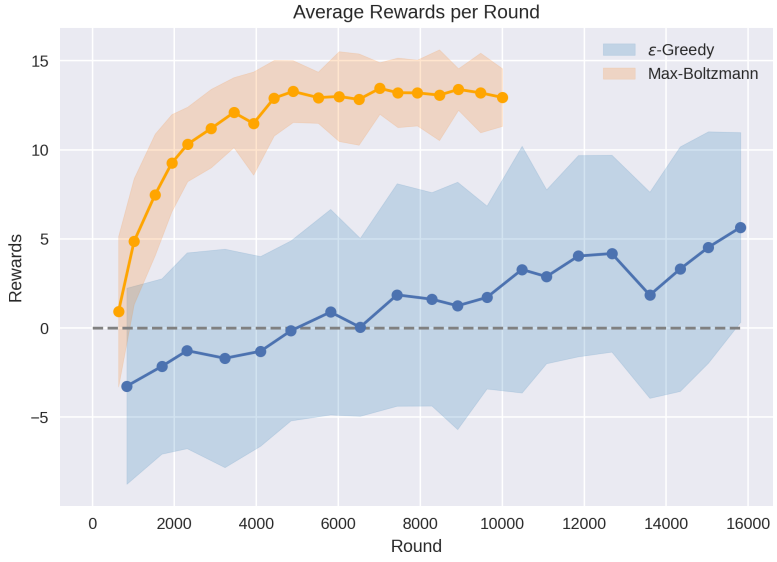

Figure 16: Example of average reward over rounds for coin collection of the ϵ-greedy algorithm compared to Max-Boltzmann exploration. As one can see, the Max-Boltzmann method achieves the training threshold within approximately 4000 rounds, while the ϵ-Greedy algorithm rises steadily but slowly.

4.2 Crate Destroying Agent

4.2.1 Q-Learning

by Leonard Marks

At an early stage it became clear that the crate density needed to be reduced for the training, as the agent started to loop very fast. That is because the agent had not much free space at the start and in combination with ϵ-greed algorithm there were not many states the Q-learning algorithm had visited. But with a sufficiently small crate-density the agent learned quite sensible movements after around 10^4 training rounds and started to destroy crates without killing itself most of the time. A lot of loops and not visited states still occurred, which is reasonable as the Q-table for the crate-destroying-agent has over $12 \cdot 9 \cdot 11 \cdot 6 = 7128$ entries. To find these loops and missing values we trained the agent for $5 \cdot 10^4$ more rounds with only exploitation, this resulted in a significantly better agent but still was not perfect.

4.2.2 Q-learning with Gradient Boost

by Johannes Pfeil

Unfortunately, we did not manage to finish the agent, that uses Gradient Boost and can destroy crates, in time. In addition to its own position case, the agent also uses poorly selected functions that we have since decided against. The agent could see one field in each direction and knew the position of its own bomb relative to its own position. Even after 300000 games, the agent could not collect more than one coin per game at a crate density of 0.5. So we knew we had to design our features differently which are now listed in chapter (see 3.1).

4.2.3 Deep Q-learning

by Maximilian Richter

Just like with Q-learning, we trained our Deep Q-learning agent to destroy crates by reducing the crate density. Only providing the direction of the closest crate was not sufficient to learn a strategy, since the agent easily ends up in a situation in which its survival can not be guaranteed. This is already discussed in more detail in section 3.1. Without petty selection of the next valid actions the agent would not be able to "see" in which direction a step will lead to a possibly save tile. As soon as the model gets additional information about free ways, the learning of an nearly

optimal policy is pretty fast ($\sim 10^5$ rounds). However, balancing the rewards has shown to be a very influential aspect of finding the right policy for this task, which has been evident through ever present loops and wrong decisions. Even though our agent gets stuck sometimes, in 90% of games our agent is capable of destroying all crates. In this stage we still ignored the combination of the different tasks for the agent.

4.3 Enemy Destroying Agent

by Maximilian Richter

Like aforementioned in section 3.1.5, as soon as a crate destroying agent is trained, defining an enemy destroying agent from the crate agent is very straight forward. We have found very good results in combat against the rule based agents without much more feature design, which has already been intensively exercised for the other possible targets. This was a surprise for us and very welcomed in our limited time to finish the project, since adapting to enemies did not require any more training.

This back-door is certainly not the most beneficial method to enforce offensive strategies, but comes with an almost perfectly defensive policy. This was also the choice for our final agent.

4.4 Final Agent

by Leonard Marks, Maximilian Richter

At the end our project we had two well trained agents, one with Q-learning and the other trained by Deep Q-learning. Both agents have been equipped with the same features, events and reward values. In an epic final battle the most efficient model was decided by fate.

4.4.1 Q-learning - my_final_agent

The Q-learning was implemented via a model-class, with the different game-states as member variables, which resulted in a low complexity of the code and thus easy rewriting of strategies, but it also took longer for the agent to train. The two Q-tables for coin-collecting and crate/enemy-destroying strategies are separate trained models and due to time issues we unfortunately only managed to hard code the conditions for the transitions between these two models. While playing without enemies the model does quite well with destroying all the crates and collecting the coins

except very few loops and self-destruction occurrences. The loop issues are solved as soon as an enemy player enters the game and forcing the agent to change the locked crate by destroying crates itself and thus changing the state of the agent. To conclude the Q-learning is a good method to train systems fast which have not too many features, for the coin collection it was a very efficient method, but as soon as enemies and crates were added, the complexity drastically increased and thus was hard to code and train, due to too many loops and undefined situations.

4.4.2 Deep Q-learning - deep_agent

Just like the Q-learning model, the Deep Q-learning agent was split into two different submodels. One agent has learned to collect all coins, the other learns to destroy crates. Both models have successfully learned to escape most bombs. Brought together, our deep agent was able to beat the rule based agent and won our internal competition to choose the best agent for the tournament with other AI agents. In the final evaluation of our models we could confirm the claims of the many supporters of Deep Neuronal Networks as function approximators in Q-learning, such that, compared to Q-learning, Deep Q-learning offers greater flexibility and potential to generalize the seen experience. However, the choice of the right hyperparameters, rewards and features is a very limiting factor when designing an reinforcement learning agent. What has been lost in time for fine-tuning the network, Q-learning can train and improve their tables.

In the final tournament, our Deep Q-learning agent has outperformed the simple Q-learning agent about a significant margin. Therefore, our final choice for the combat agent is the Deep Q-learning agent (deep_agent).

5 Outlook

5.1 Upper-Confidence Bound action selection

by Johannes Pfeil

So far, we have used the epsilon greedy algorithm to weigh the exploitation exploration dilemma. One danger of this method is that epsilon can be reduced too quickly, leading the policy to a local optimum that does not match the global optimum. A more promising approach would be to use the upper-confidence bound action selection method. Here, the action is not selected randomly as in the epsilon greedy algorithm, but the action is selected whose Upper Bound of the Confidence Interval of the respective Q-value is the highest. This prioritises exploration when one cannot yet be sure whether the Q-values are reliable and accurate where exploitation is prioritised if it is very likely that the action with the highest Q-value is also the best action.

5.2 Finish the Q-learning Agent using Gradient Boost

by Johannes Pfeil

Due to time constraints, we have not further developed the Q-learning agent that uses gradient boost. If we had had more time, we would have implemented the gradient boost algorithm for the other Q-learning agent. However, this agent works well even without the regression model.

5.3 Select only accurate Q-Values when using gradient boost.

by Johannes Pfeil

So far we have used all Q-values from the Q-table to train the gradient boost forest. Since no weighting of the data was carried out, Q-values from states that have not been visited often lead to poor learning behaviour. An improvement would be to use only Q-values with a low standard deviation to train the forest. A threshold for the standard deviation could be set as a hyperparameter. A disadvantage of this would be that initially only Q-values from states that are visited very frequently would be used. Important but rarely visited states would not be included in the training. A more promising idea would be to weight the Q-values according to their standard deviation before they are included in the training. This way, all Q-values are taken into account without initial Q-values or other inaccurate Q-values influencing the model

too much.

5.4 Monte Carlo Tree Search

by Johannes Pfeil

A promising method we would have liked to try would be the Monte Carlo Tree search. This method has been implemented in many Go or Chess playing AIs. The basic idea of this method is to very often simulate many random moves in advance to find the next most promising action. No features need to be selected and the AI does not need to be trained in advance. However, with the current implementation of the Bomberman game, we found it difficult to simulate moves in short time, so we quickly abandoned this idea. Perhaps the engine could be extended so that it is easy to simulate a light weight version of the game quickly and easily.

5.5 Exploiting the symmetry of the game

by Johannes Pfeil

By mirroring the entire playing field, an analogue valid state is created. This also applies when the playing field is rotated by 90°. Thus, seven new analogue states can be created for one state. If one updates a Q-value to a state-action-pair, one could also update all Q-values to analogue state/action-pairs. By efficiently calculating the analogue state-action pairs, the training time could be greatly reduced.

5.6 Model and sub-model

by Leonard Marks

We trained the crate/enemy-destroyer and coin-collector seperate from each other. A good way to combine these is to construct a model with different actions, like choosing which sub-model should be called, and individual rewards like Enemy Destroyed and Coin Collected, in a way so that one sub-model is not overused. With this model we can independently train certain situations and not break the well trained situation handlers (sub-models) by over-training our model.

5.7 Curiosity Driven Exploration

by Maximilian Richter

All of our exploration techniques have yet depended on an evolving probabilistic action choice (ϵ-Greedy and Max-Boltzmann, see Sec.3.3.2. One could further define a separate network chain, which gets trained to reproduce the choice of the Q-network. This can be used to assign a measure of how 'surprising' the action was, given a specific state. This measure can then be used to assign a reward for 'curiosity'. This is the main idea behind curiosity driven exploration [6], to encourage the agent to try out uncommon selection and thus, explore the space of states in a very efficient and meaningful way.

References

[1] J. H. Friedman. Greedy function approximation: A gradient boost machine. *The annals of statistics*, 29(5):1189–1232, 1999. URL https://www.jstor.org/stable/2699986.

[2] J. Groot Kormelink, M. Drugan, and M. Wiering. Exploration methods for connectionist q-learning in bomberman. 01 2018. doi: 10.5220/0006556403550362.

[3] U. Köthe. Script to lecture 'fundamentals of machine learning ', 2020. Accessed: 2021–03-29.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. Bellemare, A. Graves, M. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–33, 02 2015. doi: 10.1038/nature14236.

[5] A. Paszke. Reinforcement Learning (DQN) Tutorial. https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html, 2017. [Online; accessed 23-March-2021].

[6] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell. Curiosity-driven exploration by self-supervised prediction. *CoRR*, abs/1705.05363, 2017. URL http://arxiv.org/abs/1705.05363.

[7] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL http://incompleteideas.net/book/the-book-2nd.html.