Fundamentals of Machine Learning

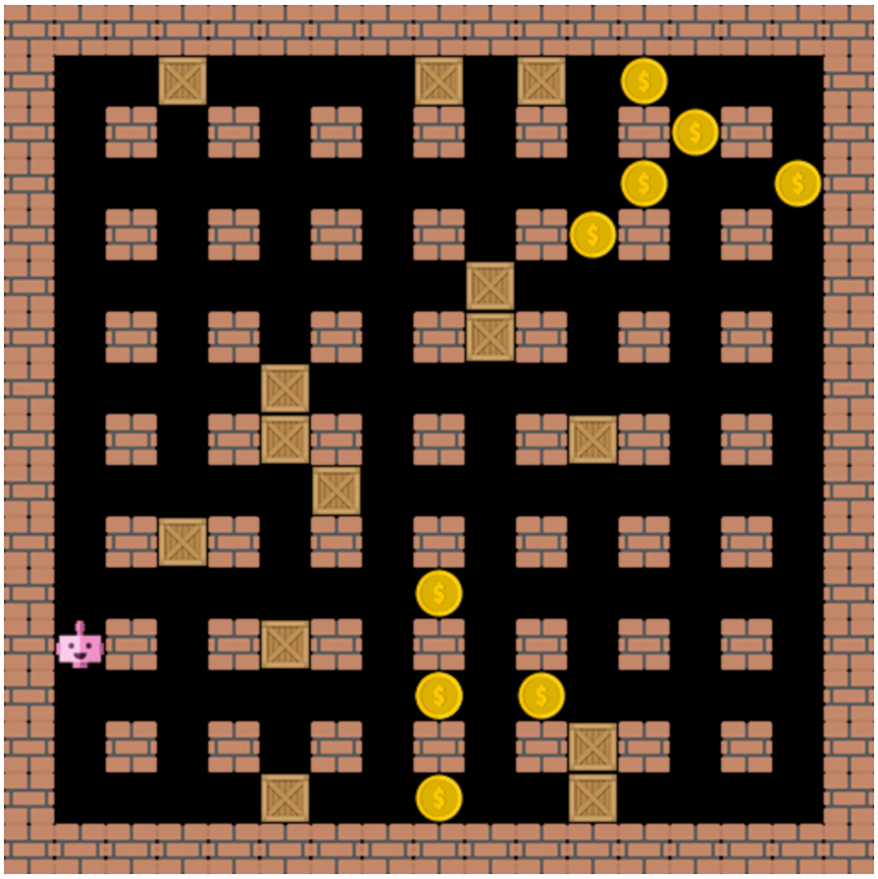
Final Project - Report

Reinforcement Learning for Bomberman

`https://github.com/hericks/FML`

Alessandro Motta, Matthias Hericks, Mika Rother

March 28, 2021



Contents

Chapter 1

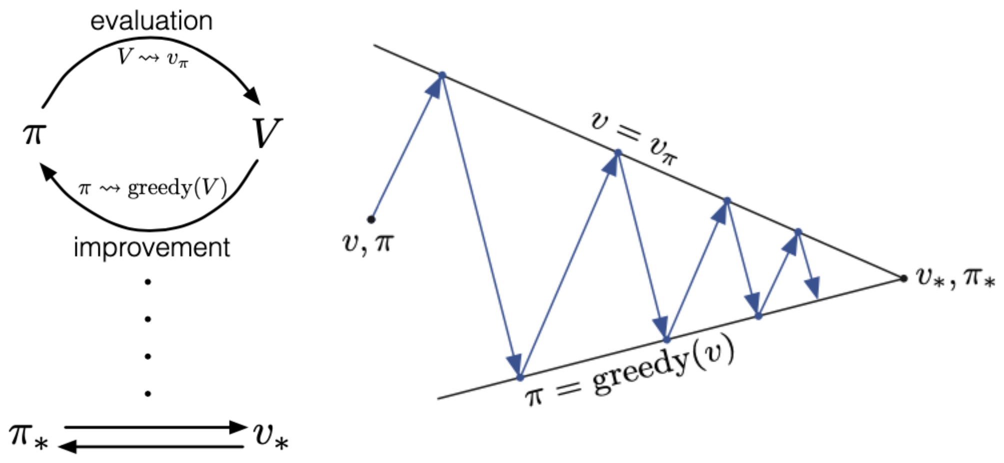
Learning methods and design choices

1.1 Learning methods

by Matthias Hericks

1.1.1 Generalized policy iteration

In their most abstract form, all of the reinforcement learning methods we deployed during this project can be described as *generalized policy iteration*. The key components of generalized policy iteration are two interacting processes. The *policy iteration* process, makes a value function v or action-value function q increasingly consistent with the policy π currently deployed. This computation of v_π and q_π for an arbitrary but fixed policy π is also called the *prediction problem*. The second main part of generalized policy iteration is the *policy improvement* process. This process makes the policy π greedy with respect to the current value function v or action value function q, respectively. Richard Sutton introduced the term generalized policy iteration to refer to the general idea of letting policy-evaluation and policy-improvement processes interact, independent of the exact details of these processes. Furthermore, Sutton describes that generalized policy iterations in many (theoretical) cases convergence to the optimal (action-) value function (q^*) v^* and optimal policy π^* is guaranteed as long as the process continues to update all states.



This generalized policy iteration is one possible solution to the *control problem* (in contrast to the prediction problem), that is the relevant problem of finding a good policy.

The policy improvement step is typically done by making the policy π greedy with respect to the most recent version of the value function v or action value function q. During the project, we relied on an action-value function q, since in this case no model of the environment's dynamics is required to construct the greedy policy

$$\pi(s) := \arg\max_{a \in \mathcal{A}} q(s, a). \tag{1.1}$$

For the evaluation or prediction part of the generalized policy iteration, we relied on on-policy *Temporal Difference* learning. Just like Monte Carlo methods, Temporal Difference methods are model free and allow for learning directly from raw experience without a model of the world. In contrast to Monte Carlo methods, Temporal Difference methods allow for *bootstrapping*, that is, they can update estimates based in part on other learned estimates, without waiting for a final outcome. This was important for the Bomberman example, since the agent should not be forced to wait till the end of a round for an update.

1.1.2 Sarsa algorithm

The first Temporal Difference learning method, we used was the *Sarsa algorithm*, which updates after every transition from a nonterminal state.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\big], \tag{1.2}$$

where

- S_t denotes the initial state of the state transition,
- A_t denotes the action taken in the transition,
- R_{t+1} denotes the reward the agent received for the transition.

$Q(S_t, A_t)$ denotes the value estimate of the state-action-tuple (S_t, A_t) and is defined to be zero for all tuples (S_t, A_t), where S_t is terminal.

- S_{t+1} denotes the state of the agent after the state transition,
- A_{t+1} denotes the action the agent will take after the state transition. Note that the value estimation $Q(S_{t+1}, A_{t+1})$ is zero for S_{t+1} terminal, independently of the action A_{t+1}. Thus, no action is required for the evaluation of terminal states S_{t+1}.

Moreover, this first algorithm introduces two important hyperparameters,

- $\alpha > 0$ denotes the *learning rate* and controls the effect of a single
- $\gamma \in [0, 1]$ denotes the *discount factor*, which is a measure for the weight of future rewards.

The update rule uses every element of the quintuple of events, $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, that make up a transition from one state–action pair to the next. In this way, the quintuple gives rise to the name Sarsa for the algorithm. Sarsa was our initial choice for a learning algorithm since it has excellent convergence properties as proven by **??** and is easy to implement.

From the Sarsa prediction algorithm it is straightforward to construct an on-policy control method using the generalized policy iteration. To do so, we gradually estimate the action-value function q_π with the behaviour policy π and simultaneously let π greedily adapt to q_π as in (1.1). One of the frequently encountered conditions to guarantee the convergence of this iteration, is that all state-action pairs are in theory visited infinitely often. Without doubt, this is not possible in the large state space of the bomberman game. Still, exploration was necessary as easily seen in the classical example of the *exploration-exploitation-tradeoff*.

1.1.3 Exploration-exploitation-tradeoff

In the beginning of the project, we used ε-greedy policies to ensure exploration of unknown action-state combinations. For every policy action-value function q and $\varepsilon \in [0, 1]$ the corresponding ε-greedy policy selects a random action with probability ε and the action greedily proposed by q with probability $1 - \varepsilon$. Thus,

$$\varepsilon\text{-greedy}(q)(s) \sim \begin{cases} U(\mathcal{A}_s) & X = 1 \\ \arg\max_a q(s, a) & X = 0 \end{cases}, \tag{1.3}$$

where $U(\mathcal{A}_s)$ denotes the uniform distribution on the set of possible actions in the state s and $X \sim \text{Bin}(1, p)$ denotes a binomially-distributed random variable with success probability p.

In later stages of the project in which it was necessary for the agent to cautiously drop bombs to complete tasks, exploration with blind tentative ε-greedy policies was not practicable anymore. Therefore, we switched to softmax-policies to still guarantee exploration. For every action-value function q and *temperature* $\rho \in (0, \infty)$, the ρ-softmax policy selects actions in state s with varying probability according to the corresponding estimated value $q(s, a)$ of the state-action tuple (s, a). To be more precise,

$$\mathbb{P}(\rho\text{-softmax}(q)(s) = a) = \frac{\exp\left(\frac{q(s,a)}{\rho}\right)}{\sum_{a_i \in \mathcal{A}_s} \exp\left(\frac{q(s,a_i)}{\rho}\right)}.$$

Similar to ε for ε-greedy policies, the temperature ρ determines the extent of the probabilistic nature of the corresponding ρ-softmax-policy. For $\rho \to \infty$ the corresponding ρ-softmax policy converges to the uniform distribution $U(\mathcal{A}_s)$ on the set of possible actions $\mathcal{A}_s$. For $\rho \to 0$ the ρ-softmax policy selects the action with maximal estimated corresponding value $q(s,a)$ with arbitrarily high probability.

A more detailed description of our decision-making process regarding the different exploration possibilities can be found in section **??**.

1.1.4 The tyranny of the single time step

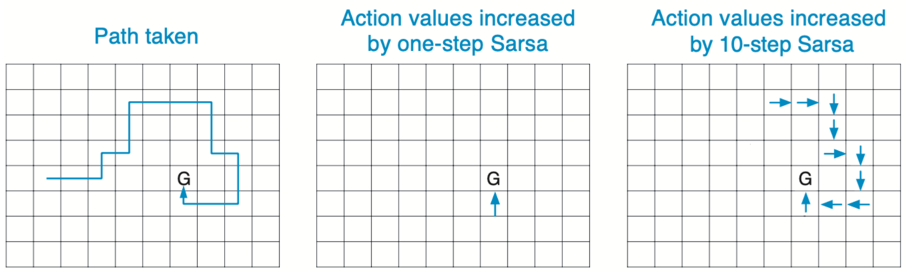
The Sarsa algorithm performed really well to train some initial agents on the first task. Still, one major drawback of single time step methods are described by Sutton as the *tyranny of the single time step*. The tyranny of the single time step denotes the disadvantage of one-step method, such as Sarsa, that the update of an action and the amount of bootstrapping is linked. It is reasonable to update the action value function frequently to take into account anything that has changed. Unfortunately, bootstrapping works best over longer time periods in which a significant change in states has occurred. For one step methods the update intervals and bootstrapping time periods are equal and a compromise must be made. To defy the tyranny of the single time step and to enable fast training, we therefore switched to an n-step method with bootstrapping over multiple steps.

1.1.5 n-Step Sarsa

The simple idea behind n-step Sarsa is to modify the update rule (1.2) to update an earlier estimate based on how it differs from an estimate *after n-step*, instead of based on how it differs from an estimate after a single time step. This gives rise to the new update rule

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \big[\sum_{i=1}^{n} \gamma^{i-1} R_{t+i} + \gamma^n Q(S_{t+n}, A_{t+n}) - Q(S_t, A_t) \big]. \tag{1.4}$$

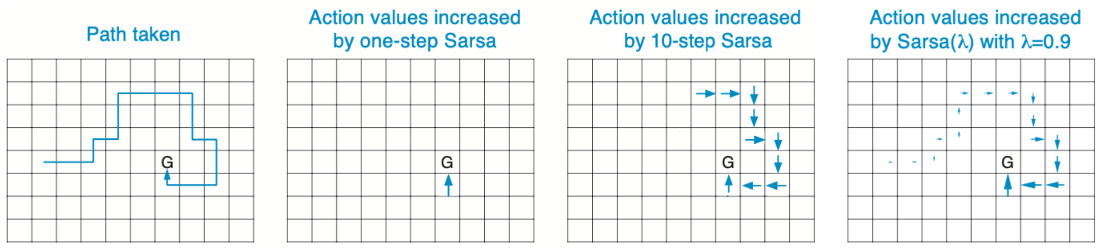
The n-step Sarsa methods form a family of learning methods. Since Monte Carlo methods perform their update only after every game, they can be considered as an extreme case of n-step Sarsa methods with $n = T$, where T denotes the number of steps per game.


Path taken
Action values increased by one-step Sarsa
Action values increased by 10-step Sarsa
G
G
G

Controlling from which states bootstrapping occurs is important, because it affects the fundamental trade-off between bias and variance of updates. Special motivation for the upgrade of the Sarsa algorithm to the n-step Sarsa algorithm was provided by the paper *Effective Multi-step Temporal-Difference Learning for Non-Linear Function Approximation* by **??**. We achieved the best training results with an intermediate value of n ($n \approx 4$), which illustrates how the abstraction of one step Temporal Difference and Monte Carlo methods to n-step learning methods can increase the performance of the two extreme methods. The results of our associated simulation study are in line with previous research by Sutton **??** and they can be found in section **??**.

1.1.6 Learning with eligibility traces

The last and final update to the learning algorithm was to upgrade the n-step Sarsa method to a generalized Sarsa(λ) algorithm. Just like n-step Sarsa, this new family of methods unifies Temporal Difference learning and Monte Carlo methods. Sutton describes the n-step approach that we have been taking so far as a forward view of a learning algorithm. For every state visited, the update algorithm looks n-steps forward in time to all the future rewards and punishments to determine the next update. The Sarsa(λ) methods reverse this view by using a short term memory, which stores information about the recently visited states and updates them accordingly. The results are similar to an n-step method, but moreover eligibility traces offer an elegant algorithmic mechanism with significant computational advantages. Instead of storing the last n state transition tuples $(S_t, A_t, S_{t+1}, R_{t+1})$, the Sarsa($\lambda$) algorithm stores an eligibility trace z of dimension d. When the action-value function is not approximated by a parametric function $\hat{q}_w$, d is equal to the product of the cardinality of the state space and the cardinality of the action space. This is infeasible for the Bomberman game with large state space. Still, we restrict ourselves to this theoretical case here to provide a clear explanation. In this setting we may index z by a state and an action $z_{s,a}$. $z_{s,a}$ serves as a short-term memory, which measures the weight of the state-action tuple (s, a) on the current state. Computationally, this is done by bumping up the component $z_{s,a}$, whenever the state-action pair (s, a) participates in producing an estimated value, e.g. action a was performed in state s. Then, the corresponding component $z_{s,a}$ fades away exponentially in time. When the agent receives a reward, all state action values $q(s, a)$ are updated, but the update is weighted by their corresponding component in z. Thus, larger updates will be performed for the tuples (s, a), which were visited just shortly before the reward occurred.


Path taken
Action values increased by one-step Sarsa
Action values increased by 10-step Sarsa
Action values increased by Sarsa(λ) with λ=0.9
G
G
G
G

The hyperparameter $\lambda \in [0, 1]$ determines how fast the components of the eligibility trace z fade away in time. The fading of the components is implemented as a multiplication of z with λ in each time step. In the first extreme case $\lambda = 0$, a new reward won't effect the action-value estimation of a state-action pair, which occurred more than a single time step ago. In particular, we have Sarsa(0) = 1-step Sarsa = Sarsa. The second extreme case $\lambda = 1$, yields the well-known Monte Carlo method, again. Thus, the family of learning methods Sarsa(λ) unifies single step Temporal Difference learning and Monte Carlo methods in a continuous way, in contrast to the discrete unification of the two families by n-step methods. Moreover, Sarsa(1) is applicable even in infinite non-episodic task, where the standard Monte Carlo method fails, since there is no point in time to perform the update.

In our case, the switch from n-step Sarsa to the Sarsa(λ) method did neither come with an increased training performance nor with a decreased training performance. Nevertheless, this was not clear beforehand. Sutton for example provides studies in his book, where Sarsa(λ) easily outperforms multi-step methods. In hindsight, we learned a lot while studying the theoretical basis for Sarsa(λ) and our final implementation is much more elegant and straight-forward than the implementation of the n-step variant.

1.2 Action-value function approximation *by Matthias Hericks*

In the previous section, no assumptions were made about the structure of the action-value function q and we implicitly considered tabular solution methods in their most general form. Because of the large feature space of the Bomberman environment, this is infeasible in practice. For the actual implementation, we had to rely on a parametric action-value function $\hat{q}_w = \hat{q}(\cdot, w)$ depending on a *weight vector* w of dimension $d \ll |\mathcal{S}|$. Therefore, a change in the weight vector potentially effects the estimation of the value of many different action-state tuples.

Fortunately, all of the learning methods presented in the last section can easily be modified to update the weights of a parametric action-value function. Generally, the weight update can be written as

$$w_{t+1} \leftarrow w_t + \alpha\big[U_t - \hat{q}(S_t, A_t, w_t)\big]\nabla\hat{q}(S_t, A_t, w_t), \tag{1.5}$$

where $\nabla\hat{q}$ denotes the gradient of $\hat{q}$ with respect to w and U_t denotes the *update target* that s's estimated value is shifted toward. In the Sarsa algorithm, the update target U_t is defined as $R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, w_t)$. Using this definition of the update target 1.5 becomes

$$w_{t+1} \leftarrow w_t + \alpha\big[R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, w_t) - \hat{q}(S_t, A_t, w_t)\big]\nabla\hat{q}(S_t, A_t, w_t),$$

which nearly resembles the Sarsa update in (1.2). Moreover, we modified the n-step Sarsa update as well as the Sarsa(λ) methods accordingly.

More concretely, we began with a simple linear function approximation, that is we write

$$\hat{q}(s, a, w) = w(a)^T \cdot x(s) = \sum_{i=1}^{d} x(s)_i w(a)_i,$$

where

- $x(s)$ denotes a d-dimensional feature vector extracted from the game state s (see Chapter **??**),
- $w(a)$ is a weight vector for every possible action a.

In this section, we have to ensure that (1.5) only updates the weights $w(A_t)$ for the specific action taken.

The advantage of using a linear function approximator is its simple form as well as the excellent convergence properties proven in **??**. This linear function approximation was sufficient for many different tasks. The disadvantage is the need for *good* features $x(s)$. In the end, we sticked with this linear model and tried to increase the quality of the model by crafting high-quality features by hand as well as with the help of a genetic algorithm for feature extraction.

Chapter 2

Training process

2.1 Initial normalization

by Matthias Hericks

2.2 Feature design

by Mika Rother

2.2.1 Simple features

First of all, a linear model requires good feature design. Since many features cannot be represented by such a model, the return values of a feature had to be either 1 or 0. Accordingly, it is not possible to create a *numpy array* that indicates for all bombs on the field how long a bomb needs to explode. In such a case, one would logically need an array for each time step of a bomb in a linear model that indicates whether a bomb will explode after as many time steps.

Accordingly, the first step was to design simple features that could be used to analyse the agent's behaviour and achieve initial training results. At the same time, it was also important that the features did not define a desired behaviour too precisely, so that the agent still had to learn to behave well itself.
s

So the work started with designing features that would teach the agent on a field without crates and only with coins to collect these coins as efficiently as possible.

Coins per quarter

The first feature that was developed was very simple: coins per quarter. The idea of this feature was based on giving the agent four features, where each of the features was for one of the possible moves, (where `'WAIT'` and `'BOMB'` were taken out as possible actions for this part, because it wouldn't make sense to place bombs or wait on a board without crates). The playing field was accordingly divided into top left, top right, bottom left and bottom right. Then the sum of the coins in each of these quarters was added up and given to the agent, so that he was always informed in which part of the field the most coins could be collected.

Training with these features alone led to some problems, of course. On the one hand, it could happen that a change of position led to a situation where suddenly the most coins were in a different quarter than in the step before. This often resulted in a back and forth movement, which is of course very inefficient. On the other hand, the agent often passed by nearby coins on the way to the quarter with the most coins. To fix this problem, we gave the agent more features.

Relative environment of the agent

Since the agent had not been aware of its immediate surroundings so far, *relative maps* were created to tell it which objects are in its immediate neighbourhood. Since only *walls* and *coins* were on the field at this point, maps were first created for these two objects. These then had the following format:

```
wall_map = np.zeros((31, 31))
coin_map = np.zeros((31, 31))
```

So both were maps of size 31×31. The idea of these maps was that the agent is always in the centre (`pos = [15,15]`) and sees the environment from his point of view. This had to include a total of 31 fields on both axes so that the agent also had a complete map available in the corners of the field.

It is easy to calculate that 31×31 entries for two maps lead to an extremely large number of features, since each field has either the value 0 or 1, depending on whether the object to be examined is located on this field or not. Therefore, and because primarily the immediate environment of the agent is interesting, another variable was introduced, the `NUM_LOOK_AROUND`. If, for example, the value 3 was chosen for this variable, the agent only got informations about his environment with radius 3, i.e. three fields in each direction from the agent and thus a 7×7 array for the relative maps. With the help of these features, considerable progress could already be made, even if the road to perfection was still a long one.

2.2.2 Escape death features

So, since the agent was able to collect coins in an acceptable time, the next step on the list was to move on to crates. Now the agent first had to learn to plant bombs to destroy crates without blowing himself up, and then collect the coins that appeared in the explosion. This task proved to be much more difficult, as the agent still had to earn as many coins as possible without blowing himself up.

Get all safe tiles

The first step was to find all the fields that could turn out to be unsafe. For this the function `get_unsafe_tiles()` was written, with the help of which one could check which fields in the periphery of a bomb were dangerous, i.e. on which fields the agent would not be allowed to stay without diying if the bomb exploded. This function was then passed the `field` and the `bombs` on each turn, which could be imported from the current game_state.

Furthermore the function `get_reachable_tiles()` was created. This function was passed the current position of the agent (`pos`), the number of steps the agent can still walk until the bomb explodes (`num_steps`) and again the current playing field (`field`). Using this data, the function could then determine which fields could still be entered in a given number of steps.

These two functions were now used to find all fields that can still be entered in a given number of time steps without dying. This resulted in the following function:

```python
def get_reachable_safe_tiles(pos, field, bombs, look_ahead=True):
    if len(bombs) == 0:
        raise ValueError("No bombs placed.")

    timer = bombs[0][1] if look_ahead else bombs[0][1] + 1
    reachable_tiles = set(get_reachable_tiles(pos, timer, field))
    unsafe_tiles = set(get_unsafe_tiles(field, bombs))

    return [pos for pos in reachable_tiles if pos not in unsafe_tiles]
```

With the help of this function it was now possible to teach the agent not to enter fields that led to a certain death. Because the function `get_reachable_safe_tiles` returns all positions, which an agent can still enter without dying.

Get safe death features

However, in order to be able to represent the whole thing as a feature in a linear model, of course only binary values (0 and 1) may be returned, so that training can be done in the correct way. Accordingly, a new function was created, which creates a numpy array with five entries (for the actions: 'UP', 'DOWN', 'LEFT', 'RIGHT' and 'WAIT' and depending on which action is feasible without dying, has a 0 or 1 in the respective entry.

For example, if the agent can only go up to avoid dying, the new function would return the following:

```python
  ret = ([0,1,1,1,1])    # (['UP','DOWN','LEFT','RIGHT','WAIT'])
```

The function that creates this output is called `get_safe_death_features`. Thus, in addition to the possibility of collecting coins, a list of features was now given to help the agent understand which fields it would be better to stay away from in order to avoid death.

Up to this point, the agent was simply trained to avoid fields that could end in his death. However, he was not explicitly told not to commit suicide. For a behavior that should be avoided in any case that the agent blows himself up by his own bombs. For example, it should be made clear to him at the beginning of the game that he must not die by planting his own bombs. This idea served mainly to ensure that the first action is not `'BOMB'`, since that usually ends in certain death. Therefore, another feature `is_bomb_suicide()` was developed, which returns whether a bomb planted by the agent led to his death.

Relative maps for escape death features

To round off the whole thing, relative maps for crates were generated, which were used to pass the crates in the immediate vicinity to the agent for training. The aim here was, of course, that the agent finds the way to neighboring crates as quickly as possible and does not have to search for so long.

As mentioned at the beginning of this chapter, in order to have all the time steps of all bombs and where they do damage, it would be necessary to create a relative map for each time step of a bomb. Since it would then be necessary to create four different maps, corresponding features were written, but rarely used for training, since it would then have been much more difficult to find good weights. And one of the goals of reinforcement learning is to train the agent with well-chosen rewards and not with a large selection of features.

2.2.3 Shortest paths

Now the point had come where our agent was performing his tasks to some extent, but not yet very efficiently. Therefore it was necessary to give him not only his environment, but actually the way he has to go, if he wants to find the next coin, for example. Since it is often important to know not only the way to the next coin, but also to the next crate or opponent, the function that calculates this shortest path had to be valid for several objects.

The first approch that was taken was a simple *breadth-first-search* algorithm that treats the game board like a maze.

BFS in a maze

In order to be able to represent the playing field as a labyrinth, a distinction had to be made in the first step between accessible fields and walls. Logically, fields with a wall or a crate were displayed as walls, the rest were displyed as free fields. This was then stored in an array (`free_tiles`) that has the same shape as the game board.

Based on this array, it was then checked if the agent has the possibility to reach its target. The parameters `free_tiles`, `start` and `target` were passed to the function `shortest_path()`. Here, the positions `start` and `target` represent the position of the agent and the position of its target. A maze was then created, looking in each iteration to see which points could be reached with how many steps. The search algorithm that was used for this process was *breadth-first-search* (BFS). If the target position was reached, the search ended and the path was reconstructed. If no path could be found, the position of the agent was returned.

Although the shortest path could be calculated using this function, there was a major problem: the maze was recalculated for each step, which had a runtime complexity of $\mathcal{O}(0.02\,\mathrm{s})$. That doesn't sound like much, but if one wants to train several workouts with at least 100 laps, then that was definitely too much.

A* search

An algorithm for finding the shortest path is the A* search algorithm. This is a generalization and extension of Dijkstra's algorithm, although Dijkstra's algorithm can also be reduced to the A* search algorithm. Now a new class has to be introduced to perform A* search:

```python
class Node:

    def __init__(self, parent=None, position=None):
        self.parent = parent
        self.position = position
        self.g = 0
        self.h = 0
        self.f = 0

    def __eq__(self, other):
        return self.position == other.position

    def __lt__(self, other):
        return self.g < other.g
```

A node is thus initialized with a parent, a position and values f, g and h. The A* search algorithm always examines the nodes first that are the most probable to lead to the goal. For this, the values for f, g and h are required. If x is a node, the following relation holds:

$$f(x) = g(x) + h(x) \tag{2.1}$$

In this context, $f(x)$ is the estimated distance from the starting node to node x, $g(x)$ denotes the previous cost from the starting node, and $h(x)$ is a heuristic that estimates how far it is from the current node to node x.

Let u and v be two nodes and let v be the target node. Then the heuristic that was used in our A* search algorithm is given by

$$h(u) = 2 \cdot (u[x] - v[x]) + 2 \cdot (u[y] - v[y])$$

where x and y describe the coordinates of the game board. The choice of the heuristic is only a small detail, which can be improved, but since later only the nearest coins are considered, this was not necessary here. Nevertheless, an alternative evaluation function is given here, which Xiang Liu et al. tested in [*] for different mazes:

$$f(x) = g(x) + h(x) + h(y)$$

where y is the parent node of x. The is equal to the definition of the evaluation funtion defined in Eq. (2.1).

The A* search algorithm then creates two lists with the open nodes and the closed nodes. Then, starting from the starting node, the neighbors are traversed and whenever a node is found that has not yet been added to the closed nodes, a check is made to see if it is already among the open nodes. If not, it will be added and its g value will be increased by one compared to the node from which it was reached. If the node is already in the list of open nodes, it is checked if the new g value is smaller than the previous one. If it is, the entry is renewed. This also ensures that the parent node is always initialized with the predecessor node from which it is quickest to get to the current node.

If a shortest path to a position is found using this algorithm, this position is returned. If not, the position of the agent is returned instead. It has been shown that this algorithm is much faster than the maze-based BFS algorithm. However, the structure of the nodes has not yet been fully exploited, so the f,g, and h values have not yet been well accounted for.

A* search with priority queue

To get the last bit of performance out of the algorithm, it was necessary to store the open nodes list as a priority queue, where the priority in this case is determined by the f value of a node. The Python module `heapq` was then used for this, which provides basics for efficient data structures. This finally allowed a satisfactory runtime to be achieved, especially since later only the objects nearest to the agent were considered. And accordingly not all objects had to be traversed.

Find shortest path to specific objects

Once an efficient algorithm was available that could calculate the shortest path, the next step was to calculate only the shortest path to the objects of a class that were really close. Given the list of positions of all crates, one wanted to include only the relatively close crates in the calculation, since for example the amount of crates with a `CRATE_DENSITY` of 0.7 would be very large and a calculation of the shortest paths to all would cost too much performance.

Therefore, a function was written that is passed the list of objects, let's take coins as an example, and then finds the shortest Euclidean distance between the agent and the nearest coin using the following nesting numpy function (`best_dist`):

```
best_dist = min(np.sum(np.abs(np.subtract(coins , pos)), axis=1))
```

But since the coin with the shortest Euclidean distance to the agent is not necessarily the one with the shortest path, one could still extend the `best_dist` with an `offset` so that the shortest path from the agent to all coins in its neighborhood with radius R:

$$R = \texttt{best\_dist} + \texttt{offset}$$

was calculated.

The shortest path functions were then used to find the shortest paths to the next coin and to the next crate. When the shortest path function was tested with the agent collecting only coins, the results were much better than with the previous features. The idea behind using this feature for the crates as well was based on the observation that when training the agent with crates and bombs, he often did not move to other crates to destroy them. By training the agent to take the shortest path to crates, much better results could be achieved.

2.2.4 All features combined

Last small feature

Since the agent had been taught during the training with all the features described so far that he had to plant bombs to destroy crates, i.e. received a positive reward for it, unfortunately the problem occurred that whenever he could plant a bomb, he also placed one, even if no crates were destroyed by it. Therefore, another feature was needed that simply indicates whether the agent is standing next to a crate or not, with the idea that this would teach the agent to place bombs more often if it destroyed crates.

Play and train with all features

Based on all the features described so far, the agent could be trained to be capable of destroying crates and collecting the coins that appeared without dying. However, this did not always succeed and even when it did, sometimes more sometimes less well (more on this below).

2.2.5 Outlook: genetic feature selection