BombeRLeman

Report for the lecture Fundamentals of Machine Learning WS18/19

Authors: Jakob Weichselbaumer, Stephan Detert, Eike Harms
Deadline: 25. März 2019

Contents

List of Figures

1 Introduction

Machine learning has emerged as one of the most promising and active areas of research in modern computer science. In this project, we demonstrate the use of algorithms from the area of Reinforcement Learning, a subfield of machine learning, to train an agent to play the classic computer game Bomberman.

Bomberman poses a challenging application for a machine learning approach - it involves a complex, dynamically changing, high-dimensional multiplayer environment too large to capture with exhaustive approaches, yet remains visually intuitive and provides well-defined game objectives. Bomberman is easy to follow and understand, but not trivial to solve.

This report aims to give a summary of the techniques employed to solve this task, the main training schemes employed, the empirical observations made, and the final conclusions drawn from the 5 weeks of work involved in completing the project.

All implementations done as part of this project can be accessed under the following hyperlink: https://github.com/stdhd/bomberman

All authors contributed equally to the success of this project.

2 Models, Methods, Design Choices

What follows is a brief discussion of the central machine learning concepts discussed in the lecture and used in the course of the project.

2.1 Reinforcement Learning (RL) - (S. Detert)

Reinforcement learning is a set of strategies for an agent to learn the behavior of an environment by interacting with it.The so-called policy determines an agent's behaviour at a specific state. The purpose of Reinforcement Learning as a sub-discipline of machine learning is to maximize a long-term objective following an optimal behaviour policy.[Szespari, 2010][p. IX]

During reinforcement learning, the agent interacts with its environment over time, issuing actions and receiving a reward signal in response. Over time, the agent learns to discriminate advantageous actions from disadvantageous actions, in order to maximize its own expected reward over the course of the game.[Sutton and Barto, 2000][p. 2]

RL strategies are distinguished by how they choose actions during training to learn the optimal policy. Some RL strategies choose the next action during learning using a continually updated estimate of the optimal policy (on-policy), while other approaches separate the policy used during learning from the running estimate of the optimal policy (off-policy).

The reward signal is an essential element of RL methods and can be interpreted as an indicator of progress towards a target. The goal is to adapt the policy to maximize the agent's expected reward over the course of a game. When entering a state, the sum of all expected future rewards is called its value. Various RL methods rely on different value definitions, potentially combining states, actions or state sequences. Another key element of reinforcement learning methods is the model - it describes the behaviour of the environment. There are model-based approaches, which assume prior knowledge over the behavior of the environment, as well as model-free strategies, which make it necessary for the agent to learn the behavior of the environment in addition to maximizing the reward function.[Sutton and Barto, 2000][p. 6]

Finally, each RL approach defines an update strategy applied to the estimate of the optimal policy learned so far. In order to eventually converge on the globally optimal policy, an agent needs to be open to discovering new, potentially better policies (exploration), while also being able to fully converge on an optimal policy when found (exploitation). Considering furthermore that in practice, administrators of RL are interested in finding good policies

with low training volume, it becomes even more important to balance both interests against each other. [Sutton and Barto, 2000][p. 3]

2.1.1 Monte Carlo

A common family of RL techniques are Monte Carlo methods. The learning process of these methods is based on feeding the algorithm with sample state transition sequences and the total reward after following the chain. The state value function is estimated by attributing the total reward after a state transition sequence to all states in the chain reduced by a discount rate. Over time, the values of the intermediate states converge to a value function which can be used to generate optimal state combinations, starting from any state. This approach requires knowing the behavior of the environment as a model: Starting from a certain state s_0 and taking into consideration the environment model to determine all possible next states allows estimation of the most valuable following state s_1^* [Sutton and Barto, 2000][p. 91]

A different way to implement the Monte Carlo approach is to estimate the state-action value function $q(s, a)$ if there is no model of the environment behaviour available. As the value of single states was estimated in the model-based approach, under the model-free approach, future returns of state-action pairs following a policy afterwards are estimated. For each state, the agent chooses the highest $q_\pi(s, a)$ applying the policy π. [Sutton and Barto, 2000][p. 96]

2.1.2 Q-Learning

Besides Monte Carlo, there is the approach of temporal difference learning. Instead of updating the current estimation of state values after the return of the episode is known, temporal difference methods update the value estimation after any action is taken - there is no need to provide full valid episodes as training data. [Sutton and Barto, 2000][p. 120]

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \right] \tag{2.1}$$

In temporal difference learning, the value update is performed immediately after every state transition with the reward R_{t+1} and the value estimation $V(S_{t+1})$. Monte Carlo and TD methods both converge asymptotically towards the optimal policy - still, the advantage of TD is obviously that there is no need to wait for the completion of a whole episode before the value function can be updated. [Sutton and Barto, 2000, p. 124]. The team focused on temporal difference learning in the final implementation, as this step-based rewarding allowed observing the learning process in a much more precise way and simplified debugging tasks.

A simple temporal difference learning algorithm is Q-Learning. This algorithm approximates the action-value function Q, estimating the value of an state-action pair with help of the maximum state-action value $max_a Q(S_{t+1}, a)$ of the following state. All elements needed to implement a Q-Learning algorithm are policy, reward signal and an update rule (see 2.2). As Q-Learning is an off-policy approach, there are few restrictions on the policy used to generate training data. Convergence of the state-action value function Q is assured if all state-action pairs are asymptotically updated infinitely often.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{2.2}$$

When it comes to testing an agent equipped with a learned policy, it can be observed that for frequently visited states, learned Q-values are accurate indicators for state-action combination values. For a policy to work in practice, however, it also needs to make accurate value estimates in states it has not encountered as frequently during training. In order to achieve useful guesses for state-action values in these cases, there are techniques of approximating the Q function to generalize its applicability to novel states.Sutton and Barto [2000][p. 67f.] For closer discussion of this approach in our implementation, see 4.2.2.

2.2 Requirements - (E. Harms)

Before beginning with development of the agent, engineering suitable features and designing rewards, the team focused on analysing the requirements.

1. **Decisions made faster than 0.5 seconds**

   One hard requirement of the tournament which each agent finally will have to play is that decision taking time must be faster than 0.5 seconds. This means one has to consider implementation efficiency and sets a limit on how complex precomputations can be, e.g. for creating features.

2. **Handle unknown states as well as possible**

   The bigger the feature space becomes, the more possible observations can occur. It might not be possible to learn all of them even with long training times, which requires novel approaches for estimating action values in new situations. For discussion of training limitations and the performance of Q approximations, see chapter 4.

3. **Keep training time reasonably short**

   Considering the time limitations of the project and the need to be able to implement and evaluate changes to our model quickly, it should not take too long for agent training to produce tangible results. Training times less than a day might be fine,

but anything longer would make evaluating new updates to the model too laborious. One method to shorten training times is to practice dimensionality reduction and hand-craft expressive features for short training times; this is discussed in chapter 4.1.

4. **Learn the best decisions from observing gameplay**

   To win the tournament, the agent needs to survive as long as possible and maximize its score by collecting coins and killing enemies. However, it is not immediately clear how to judge whether an ingame decision leads to a favorable result. Before beginning with engineering work, the team had to settle on a general design direction - which class of algorithms would allow our agent to update its "knowledge" most efficiently?

2.3 Feature Modelling - (E. Harms)

When discussing the features in the beginning of the project, many ideas came up which later on were rejected again. The experience made is that it is essential to have powerful precomputed features rather than just simple information, e.g. the bomb's locations which can be taken from the game state environment. Also an important factor is the number of feature combinations which should be kept as low as possible. This can be achieved by switching off features in situations when they are not needed. A good way to find feasible features step by step is to solve the subtasks of a coin-collecting-agent and a coin-collecting, creates-bombing-agent, separately.

2.3.1 Final selection of features

Closest coin direction

In order to find coins, a breadth-first search is used to compute the next direction to the closest coin. The direction tells the agent either to go left, right, up or down. In case no coin is visible on the field this feature is switched off. It also is switched off, if there is a coin which is not reachable by this agent so that it rather focuses on objectives in his area. This becomes relevant if there are enemies which uncover coins on their side of the field. With only this feature the agent learns after a few minutes of training to collect all coins as there are only 5 observations to learn.

Best bomb field direction

This feature tells the agent which way to go to efficiently place bombs on the field. This supports it finding coins quickly in the beginning of the game. As every explosion affects a radius of three units into each direction (as long as there are no walls), a single bomb should maximize the number of affected crates while keeping the number of moves required

to get to this spot as low as possible. First of all, the distance to all reachable fields on which a bomb would hit at least one crate is calculated. The value of a field equals the number of crates a bomb on it would destroy. In order to evaluate the best field its value is divided by its distance to the agent's current position and then the maximal value selected. Using this strategy the agent finds the most valuable field in his area without walking too far.

Closest safe field direction

With the closest-coin and crate feature the agent still kills itself. It does not know where danger zones occur when bombs are placed. To indicate a direction to go in order to leave a danger zone a breadth-first search is done which looks for the closest reachable safe field which is neither threatened by a bomb nor covered by an explosion. While analysing this feature the team found out that sometimes the agent walks into a dead end. This happens because of explosions which change the reachable environment temporarily. If a safe field is not reachable because of such an explosion, the breadth-first search outputs a direction which leads the agent to any target even if then the agent ends up in a dead end. To mitigate this problem the search for a safe field is run a second time without explosions considered, if the first time it does not find a reachable target. Furthermore, the feature is switched off, if the agent is not on a danger zone because then it should rather focus on something more important than escaping.

Dead end detection

While the safe field feature does a pretty good job, still the case occurs that the agent traps itself in a dead end. The assumption is that it may learn to listen to the safe field direction but also experiences that walking on a dead end not immediately is a bad thing. To really make sure that it does not trap itself and follows the safe field the dead end detection is implemented. It is a binary feature which most of the time is switched off, so it only slightly impacts the number of combinations. When the agent places a bomb and one of the surrounding fields leads into a dead end it is activated. Observing the agent's training one sees that this feature improves the dead lock behaviour further.

Safe to go left/right/up/down

The closest safe field direction is still not sufficient to avoid suicide. The agent may know how to walk out of a danger zone, but it has no indication that hinders it to walk back in. What happens quite often is that after the bomb explodes, it walks into the still existing explosion. Hence, four binary flags was introduced which indicates whether it is safe to go left, right, up and down. The flag is zero if there is either a wall, crate or explosion.

Has bomb

This feature helps the agent to learn that it is not able to place a bomb if it has already done it and his bomb has not exploded yet. Without it it causes invalid actions which are punished, but does not learn why it is punished.

Closest enemy direction

When enemies are added to the field, the game can be split into two phases. The first one is bombing crates and finding coins as fast as possible. After crates are destroyed and coins are collected the second phase begins where the focus is on hunting enemies and trying to kill them. To distinguish the phases the feature is only switched on when there are less than 11 crates on the field. With only one available bomb it is not easy to strategically kill an enemy. The team observed that in the second phase there is much chaos when several agents are in one area placing bombs. As the team ran out of time to find a more sophisticated feature to teach the agent in which direction the closest enemy is by performing a breadth-first search. Thereby, it should join the chaos, place bombs randomly and hopefully here and there kill an enemy. With this strategy our agent should have good surviving skills.

Enemy in bomb area

This feature improves the agent's behaviour in the second phase. When an enemy is in a five field radius during that phase it is switched on, so the agent gets an indication when it is reasonable to place bombs.

2.3.2 Rejected feature ideas

Radius

One of the very first ideas was introducing a radius feature which includes a one or more field radius around the agent. It holds information about coins, explosions, bombs, crates, walls and enemies which are in one of the radius fields. However, as outlined in chapter 4.1 a sample radius of three fields which means a 7x7 matrix drastically increases the number of combinations which causes training to take much longer until improvements in the agents behaviour can be observed. During training it was noticed that also a radius of one (3x3 matrix) massively increases the number of feature combinations. Hence, it was decided not to use a radius. Most of the radius fields' information is already in some form contained in the other features, anyways.

Closest crate direction

This feature tells the agent in which direction it should go to get to the closest crate. Similar to the closest coin direction it is switched off, if there are no crates on the field. In this case the agent can focus on either collecting remaining coins or hunting enemies. Again a breadth-first search precomputes this feature. When the agent stands right before a crate this is indicated. This way the agent is able to learn when to place a useful bomb which definitely destroys crates. The feature is quite effective, but while observing training it becomes obvious that there is a more efficient way to place bombs. That is why this feature was replaced by the best bomb field feature which leads the agent to a more valuable field to place his bomb.

Closest coin distance

The closest distance to a coin was an additional feature meant to help solve the coin collecting subtask. An action which reduced the distance is rewarded. The added value of this feature, however, is not high as it does not indicate which action might be useful. There the coin direction does a better job. Hence the team discarded it.

2.4 Reward modelling - (E. Harms)

Besides selecting good features, choosing the right rewards is also a tricky engineering task. The easiest way is to just reward and punish the obvious events. For the coin-collecting subtask this can be collecting a coin, suicide and invalid actions. With these rewards the agent eventually learns a strategy to find all coins. But it takes lots of training episodes because the less rewards are given the longer the agent needs to find the optimal policy. Hence, reward shaping can be used to speed up the learning process. Rewarding the agent if it for instance follows the closest-coin direction feature converges very fast to the optimal policy. The more complex the environment though, the harder it gets to choose the right rewards. With too many rewards the agent might learn unintended behaviour.

The most important behaviour to learn is bombing crates while not committing suicide. Also dodging bombs from enemies is essential. Therefore, the action which follows the direction to the next safe field is rewarded. Furthermore, the agent gets punished if it walks into a direction which is marked as unsafe by the safe-to-go-left/right/up/down feature. By default every move is also slightly punished, so that the agent tries to optimize the number of moves. Like walking, waiting is usually punished as well except for the case it makes sense which is when there is no safe field reachable and no safe way to go. To learn where to place bombs bombing is rewarded when the bomb is on a valuable field or when an enemy is in the five field radius during the second phase of the game. Otherwise it is punished. Generally, suicide and invalid actions are punished whereas collecting a coin is rewarded. Besides that, rewarding when following the best-bomb-field and closest-coin feature lets it

focus on getting more points. The rewards for not dying should have a stronger impact than the ones for focusing next targets.

It was also discussed giving coarse rewards only for collecting a coin, killing an enemy, dying and invalid actions. However, with the designed feature set those rewards would in some cases reward the wrong action. Taking killing an enemy as an example. The reward for placing a bomb at a position which kills an enemy would need to propagate four observations backwards over several training episodes because of the bomb timer. But therefore the observation to reward should in all cases have only one best action. When only having direction features it is very likely that the observation in one case means placing a bomb but in another walking up. Hence, the reward for killing an enemy is zero.

2.5 Implementation - (E. Harms, S. Detert)

Basically, the agent code which will be submitted for the competition with other teams is required to react on the given game state within a limited time amount and return the action which is derived from the agent's policy. Different approaches in using policies to make decisions during the training process of the Q-Learning agent required on the on hand well-structured code framework, re-using as much as possible while keeping the implementation flexible in order to make short implementation and test cycles possible. Above all, while testing new features and changing the corresponding derivation functions, the so far created training data and other feature derivations must not be influenced - the code framework has to stay in a consistent state.
Facing these challenges, the following design decisions were made:

1. There is a central feature class, used by all different learning approaches to keep consistency.
2. It should be possible to pause training and resume it, again.
3. The Q-tables and observation collections have to be to be identified uniquely based on the feature configuration and can be used by all of the Q-Learning implementations.

In terms of Q-Learning the Q-table row which fits the current observation needs to be found comparing it to the learned observations. Performing the Q-Learning process as well as executing a learned policy requires frequently wrapping up information about the current game state in the way given by the team's design choices. Each game state as the total amount of available information about the game arena, the agents' states are reduced to a observation vector consisting in different features. These observation vectors are used within the development process and thus are changed often. During the development process it can be observed that deriving observation vectors is time consuming and limits the number of training rounds in a limited time period significantly. In order to overcome this issue

the team creates the class observation object implementing all functionalities to generate observation vectors the most efficient way.

2.5.1 Code framework - custom enhancements

Later will be described how the two different Q-Learning algorithms in this project were implemented. This section will be about the basic mechanisms both implementations have in common. The observation object is the common platform of the code to create observation vectors by any agent implementing Q-Learning. The constructor of this class takes the configuration parameters of the observations used in the agent as the user defines it: The radius of the observation's view of the game arena as well as a list of more additional features are given by the agent using the observation object.

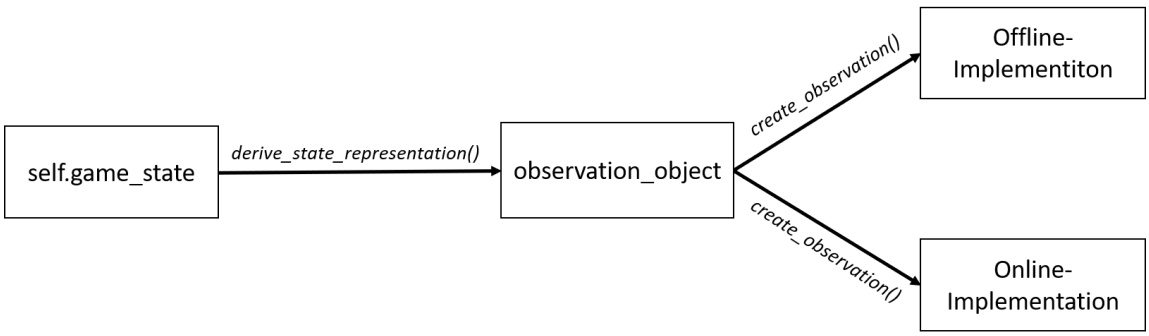

Figure 1: Use of observation object

As figure 1 illustrates the *derive_state_representation* function transforms the information given by the *game_state* into a new format. This format can be used either offline or online. The respective agent implementation calls *create_observation* to get an observation containing the computed features.

2.5.2 Feature implementation

In every game step, an object is derived from the observation object class containing all available information from the Bomberman environment. Depending on the feature configuration, for all four agents observation vectors are derived. The derivation schemes are implemented independently in callable functions within the class observation object. Building the observation during learning and testing is done by filling the empty vector with scalar values - one value per observation feature. As the observation consists of the excerpt win-

dow from the agent's surroundings and additional hand-crafted features as well, there are two steps to generate the vector:

Filling the radius map

In one step, the agent's surrounding items like coins, wall elements, crates and other agents and explosion areas are extracted from the game state, provided by the Bomberman environment class. These items are encoded uniquely as natural numbers ensuring the observation features allow the agent to learn and inserted into a two-dimensional array. The resulting array represents the view of the agent's current position on the game map. As items on the game map can overlap, developing an item hierarchy determining which element overrides each other is crucial. When the creation of the array is finished, the result is flattened to a one-dimensional vector and inserted into the observation vector.

Providing additional features

A useful example for additional observation elements is a feature guiding the agent towards the closest coin. As the score of the agent and subsequently its ranking after the tournament will depend on the number of collect coins, a suitable method needs to be found to let the agent learn to find coins taking the shortest path. As there are crates, enemies and walls on the map as well, navigation just using standard Manhattan navigation would not be useful. Hence the path finding algorithm Breadth-First search is employed for this. All coins' locations on the game map are considered targets by the algorithm while exploring all possible paths starting in the current location of the agent. The optimal path extracted by the algorithm is in a next step used to determine the best action, the agent should take to follow this path. The result is finally mapped to a natural value in $[0; 3]$ representing one of four possible moving directions. The special cases which need to be considered here are the absence of any coins and the problem that the remaining coin cannot be reached due to obstacles. As these special cases lack of a direction indication, they need to be treated a different way as will be stated, later on in this chapter.
In addition to the feature indicating directions towards coins, there are more useful features using breadth-first search: Whenever an agent is threatened by a bomb, the closest safe field direction feature indicates the shortest path towards a safe location. This feature is implemented by simulating the explosion radii of all bombs currently placed on the game map and declaring the affected locations as obstacles to the path finding algorithm.

Breadth-First Search

The breadth-first search is an important element when calculating features which point into a direction. In the beginning of the implementation phase simply the already implemented function look_for_targets from the simple agent code was used to e.g. find the direction to the closest coin. With more complexity discovering when trying to solve the subtasks it

became obvious that this was not enough. One main problem is that the original implementation does not indicate whether a target is reachable. In case it is not it just outputs a field to any target. Thus, the function was modified because the coin and the safe field feature shall be switched off when no target is reachable. Another modification was needed because of the best bomb field feature. Here, first it is evaluated which targets can be reached and then the field with the highest step-distance value returned.

Expanding the number of trained observations

Deriving the observation per each game step the naive way leads inevitably to a large number of training games to learn all possible configurations of the game state. The size of needed training data increases by every feature added to the observation or by increasing the size of the window which is visible to the agent's observation. Subsequently, it is useful to find strategies to reduce the number of training games to reach an acceptable agent performance.
The most obvious strategy which does not cause any loss of information while still decreasing the amount of necessary training data significantly is rotating the observations and the corresponding actions in the Q-table in a correct way.
Doing these rotations requires custom transformation procedures depending on the possible features included into the observation can attributed to one of the following categories:

- Space matrix: View of the game arena
- Direction oriented boolean: Criterion is true for left, right, up, down
- Value describing distance, not position related states

The number of learned observations can be expanded by rotating and reflecting the observation vectors the agent comes across within the training rounds. Features attributed to the first two groups need to be modified doing these transformations while the third group would not be changed during the process. More non-spatial features like the information, that targets such as coins are not reachable must not be rotated, as well. In general, there are eight rotations possible assuming there is no simetry in the observation. In case there are simetries, the number of unique transformations generated from the originally observed features might be smaller. Though, adding eight observations two the observations database instead of one is just one of two adaptions necessary to be applied to the regular Q-Learning algorithm. The second adaption is to transform the original action into the rotated action fitting the rotation of the direction features.

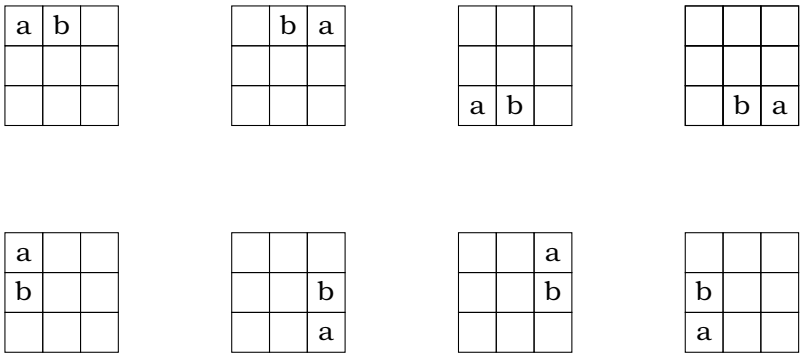

Figure 2: Transformations of a sample 3x3 game arena excerpt.

An observation, which is encountered the first time while learning is rotated according to the samples in figure 2. The resulting seven new observations plus the original observation are now added to the observation database, before the eight rows in the Q-table are processed by the update rule. Whereas the view can be transformed by using standard numpy functions on a two dimensional array, updating the Q-table is a more complex task: The update is performed on all eight Q-table rows with the same equation, but the actions for the transformations are not equal, but need to be transformed, too. To achieve this, that the spatial orientation of the axes is transformed in accordance with the transformations of the directional features.

2.5.3 Implementation offline Q-Learning

Training a Q-Learning agent is generally time consuming - as this procedure is regularly done a several times in the course of the development and testing of new features to solve the Bomberman problem, a Q-Learning algorithm is created, different from the common approach: The actions and state transitions directed by any Bomberman agent are captured and saved in numpy files. These files contain all information required to learn a Q-table from games, which have been played in the past. Building a big database of games allows the developers to test and evaluate new features and combinations very fast as there is no need to train only from slow live episodes. In the implementation of this algorithm, game states within an episode are read from the numpy file. The actions of all four participating players are assumed to be done by the learning agent, all updating the Q-table equally. As mentioned in **??**, the observation vectors are derived from all four agents' state information and then added to the observation table. Besides the Q value of the corresponding observation and action performed in the prior game step, for all four agents the reward is derived from the game state information as it contains the effects of agents' actions like "coin collected" or "agent killed". The rewards for these events can be changes before a training session like the configuration of the observation vector can be changed.
The intention to use a database of recorded games make an implementation necessary,

which learns only once with every training datum - yet training from a large database make it difficult to train on all recorded games in one session. The solution is here, that all games, which have been learned for the current training setup are marked, so the training can be paused an resumed continuing updates on the same Q-table version.

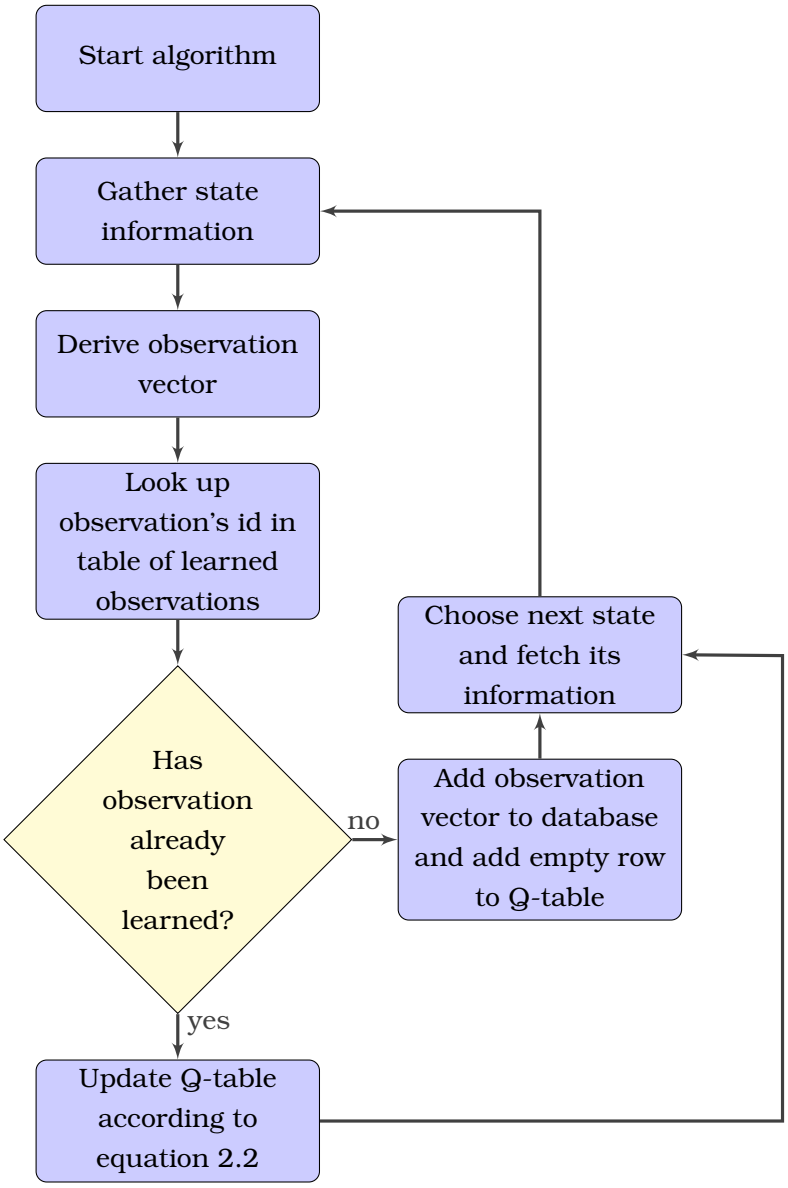

Figure 3: Flow chart diagram showing the general learning algorithm

2.5.4 Implementation online Q-Learning

The Bomberman framework provides an interface which can be used to develop an own agent. The agent's code (james_bomb_training) which comprises training and testing can be accessed on Github. The important functions are *setup*, *act* as well as *reward_update* and *end_of_episode* when the agent is in training mode.The order of called functions in training mode is illustrated in the following pseudocode.

Algorithm 1 Function calling order

```
setup()
for i = 0 to max_steps − 1 do
    State_i
    act()
    State_i → State_{i+1}
    reward_update()
end for
State_{max_steps−1}
act()
State_{max_steps−1} → State_{max_steps}
end_of_episode()
```

Setup Function

In the setup function the needed learning parameters like learning rate and discount factor for the reward update function are set as well as parameters to lower the learning rate or epsilon during training after a specific number of steps. Also, it is decided how to apply the epsilon-greedy strategy where the value of 1 means that every action is chosen randomly. Furthermore, the observation object is initialized with all relevant features. As last step the observation database which holds all known observations and the corresponding Q-table are either created if they not yet exist or read from the current folder.

Act Function

The act function consists of the testing and training part. If testing is switched on the current observation is created, then looked up in the observation database and if there is an entry the corresponding entry from the Q-table is read. From the Q-table entry the action with the biggest value is chosen. If there are two biggest action values, one of them is randomly selected and set as next action. In case the observation is not yet known in the observation database, simply a random action is chosen.

If the agent is in training mode the occurring observation is also looked up in the observation database to decide whether to append it or to update an existing entry. Afterwards, its

rotations are created. Rotations are used to speed up training by increasing the number of observations learned in one step. When the original observation is not yet known its rotations including itself have to be appended to the observations database. As the set of rotations can have duplicates those have to be eliminated first. It is important to remember the added observations' indices for the reward_update function later. Also a random action is chosen because there is no experience for this situation. If the observation already exists, only the indices are saved and the biggest action value from the corresponding Q-table is selected.

Another important decision taken here is whether to apply the epsilon-greedy strategy i.e. whether to choose a random action. Depending on epsilon set in the setup function a random action is chosen. Appending unknown or looking up known observations and saving their indices also happens when the epsilon-greedy strategy is applied.

Reward_update function

The reward_update function basically runs the Q-table update function, described in chapter 2.1.2. The corresponding equation requires the action value of the old observation, the best action value of the new observation and finally the reward.

The saved indices including all the rotations from the act function are used to fetch the old action values which also have to be updated. In order to get the best action value from the new observation, it is looked up in the observation database and its index is then used to reference the Q-table. Afterwards, the maximal value from the Q-table entry is selected. The reward depends on the observation from the previous step and the consequential action. By events it is tracked what actually happens to the agent. When e.g. standing next to a wall and then choosing the action which leads towards the wall this results in an invalid action event. The reward is the sum over fitting observation action combinations.

When the agent walks into a direction it gets the following rewards for:

Waiting is only rewarded when it makes sense. Otherwise it is punished.

When the agent drops a bomb it is rewarded when

There are a few events which are simply rewarded or punished independently of the observation. These are:

With these rewards given and the above mentioned features the agent learns already after around 2000 episodes of training to play reasonably well against simple agents.

Another crucial point which is relevant in the reward_update function is how to handle the rotations. The reward for rotations is the same as for the original observation. Thus, it has to be calculated only once. On the contrary, there might be duplicate observations. As a rotation simulates visiting the state while actually being only in the original state, for instance a two times occurring rotation causes also two updates of its action values.

Reward	**Condition**
+ 800	Following direction safe field
+ 400	Following closest crate feature, safe to go is on and closest coin feature is off
+ 600	Following closest coin feature and safe to go is on
- 600	Not listening to safe to go feature and safest field feature is off
- 50	Walking (every time)

Table 2.1: Rewards when choosing direction

Reward	**Condition**
- 500	By default
+ 600	All safe to go features are off and no safe field is reachable

Table 2.2: Rewards when waiting

Reward	**Condition**
+ 700	Best bomb place feature says it is a valuable field
+ 700	Enemy in bomb area feature is on
- 300	Bombing (every time)

Table 2.3: Rewards when placing a bomb

Reward	**Condition**
+ 500	Coin found
- 500	Killed self
- 1000	Got killed either by itself or enemy

Table 2.4: Rewards for general events

End_of_episode function

The end_of_episode function which is finally called in the framework first of all calls the reward_update function to reward the last chosen action. Afterwards, it saves the updated Q-table and observation database onto the file system. Furthermore, this function is used to lower epsilon and the learning rate during training after a specific number of steps.

3 Training process - (J. Weichselbaumer)

The following section details one of our two main approaches for implementing Q-Learning - in this case, from stored data. The team simultaneously worked on a conventional "online" approach, whose implementation is described in chapter 2.5.4

3.1 State representation

In order to test the efficacy of various machine learning approaches for our task, it became convenient to have a database of played games to use as a basis for training.

The alternative to this approach - playing games and training models "live", without saving them to the disk somehow - has the disadvantage that all games played are "lost" and cannot be reused, and that one cannot compare the performance of different models to each other on the same data. Training this way also avoids the file access conflicts that might emerge when accessing the same Q Tables or databases for multiple agent instances during a single game.

For these reasons, the team chose to develop a vectorized representation of the game state which contains all information for a single game step. This implied a crucial design measure: Construction of observations would need to be based on the state vector representation, so that training from state vectors and live execution of the agent are based on the same framework. This was not without its difficulties, as explained below.

A successful state representation is space-efficient and contains all game-characterizing events. To simplify the development of the format, we assumed that certain game parameters would remain immutable: For instance, that players could keep at most one bomb in inventory and that wall squares would be distributed in a characteristic checkersboard fashion. Other game design features, such as the size of the board,the number of players, or the number of events that can occur, do not affect the representability of the game state. In the following discussion, a standard board format of a 17 x 17 grid is assumed. The state representation takes the form of a numpy integer array and contains 3 separate parts:

The first is the board state, which consists of 176 potentially navigable fields and 113 wall fields. In order to remove redundant information, the state representation only notes the contents of the navigable fields. Board state fields contain information about: Crates, Coins, Bombs, and Explosions.

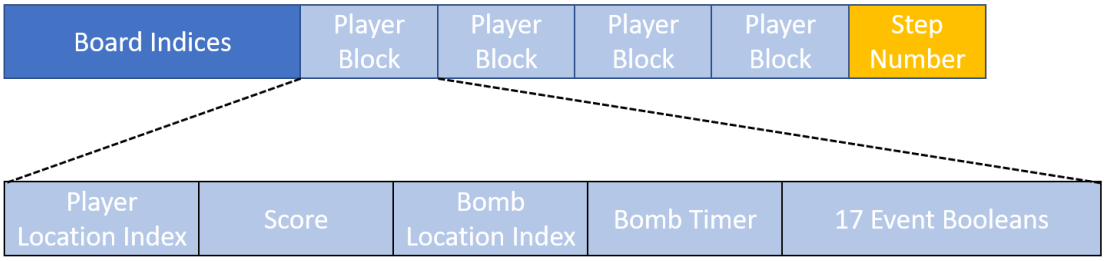

Figure 4: State vectors are split into sections to describe an entire game step. Below: Contents of the player block

To represent the board state, each navigable field is uniquely associated with an index in $[1, 176]$. Indices count through all x values in one y column of the board matrix, skipping all wall fields, before advancing to the next y column. Index 1 is associated with board coordinates (1, 1), Index 2 with board coordinates (2, 1), and so on. Crates, coins, and explosions are encoded at their respective positions in the state vector with characteristic natural numbers.

Because the game provides location information on basis of x, y coordinates, it was necessary to write functions to translate between the two coordinate representations. Initially, we had implemented these functions to arithmetically translate between index and x, y coordinates when called, but run profiling revealed that these function calls were taking up the bulk of training time. We then rewrote the functions to provide indices by accessing an imported numpy array, which significantly improved performance at runtime.

The next segment consists of four "player blocks". These contain: Information about player and bomb locations, bomb timers, player score for the current game, and the events that were passed to each player by the end of a game state. Players are written into the player blocks in the order established by the environment attribute "players".

Dead players and bombs in inventory (not on board) are noted with the board index "0". Bombs in inventory have timers "0".

The game provides information about 17 distinct events. For each player, a block of length 17 keeps track of the occurence of each game event for this player in the current game step. Events are noted in the order they occur in settings.py and are noted according to their multiplicity in the current game step. Events that do not occur are noted as "0".

Our initial approach was to note each event as a boolean, but this had the crucial disadvantage that, during training, one could not give multiple rewards for an event that occured multiple times, such as when one player destroys multiple crates in a single step.

Finally, the current step number is noted as a single integer in the last position of the state vector. In total, the state vector contains $176 + 4 \cdot 21 + 1 = 261$ integers to define the game state.

In order to generate bulk data for efficient training later on, it proved helpful to manage game state saving from the game's environment file. The original game files were adapted in a few ways to allow for effective data generation.

3.1.1 State saving from environment.py

To implement automated state capturing, a new function *capture_state* is called at the end of *do_step*, which constructs the state vector for the newly completed step and appends it to a running list of steps for the current game. *capture_state* loops through all players, bombs, coins, and explosions and saves them to the state representation vector, as well as noting the presence of crates from the arena.

Additionally, handling of "invalid actions" was augmented to note both the fact that an action was invalid, as well as the action the agent intended to take. This way, it was possible to apply a negative reward to the specific observation and action combination which triggered the event "invalid action". Before, only the fact that an invalid action was performed was noted.

At the end of the game (when *end_round* is called), the list of saved states is converted to a numpy integer array and saved to the disk using a timestamped filename defined during intialization of the environment. The "round number" is appended to the filename, allowing a script to run an arbitrary number of games and save each as a distinct file (denoting a separate game) with common timestamp.

3.1.2 State creation in callbacks.py

The main drawback of the above approach for data generation is that at execution time, agents do not have access to the environment, so there is no way to generate a state representation as used in training.

To circumvent this, the team was forced to make the design choice of implementing a second, equivalent state saving function *derive_state_representation* that is accessible to agent code at runtime, which constructs the state vector from the attributes available in the dictionary *game_state*. It proved to be a challenge, during development, to verify that both functions behave in equivalent ways.

Complicating this process is the fact that it is not possible to derive which bomb belongs to which player from the information in *game_state*. While this can be compensated for by simply assigning dropped bombs to players on the basis of which players are not holding a bomb in inventory, self-destruction leads to a major loss in score during play-time, meaning that the inability to discriminate own bombs from enemy bombs in-game can be potentially costly.

In this function, the state vector is constructed in the same way from the lists of game objects available to the player, except that, as explained, bombs are written to player blocks only to satisfy the condition that every player either is holding a bomb in inventory or (exclusive or) has deployed a bomb to the field. This would be relevant for a feature which, for example, indicates whether the closest enemy is holding a bomb.

Additionally, the player calling the state derivation function is always indexed as player 0 in the game state (player block 0), which is used later on to create this player's observation (see 2.5.1).

3.2 Data generation

3.2.1 Data production

In order to practice reinforcement learning effectively, it is necessary to have access to a wealth of high-quality data. The classical difficulty with pure reinforcement learning, however, is that as the number of features and their combinations increases, the amount of training data needed to adequately converge on correct Q values for most observations grows astronomical.

Fortunately, there are tricks available in the particular case of Bomberman to increase the amount of useful data deriveable from a single game.

The first is immediately apparent in consideration of our state-saving approach: Rather than using a single agent to train, one can use the behavior of every player in the game to learn Q values. A single game, therefore, delivers up to four episodes, namely one for each participating player.

The second way is to exploit the symmetrical nature of Bomberman: A mirrored or rotated game state produces another valid game state. In total, one can find eight such transformations (see chapter 2.5.2).

Combining both approaches, a 100-step game gives us up to 3200 distinct observations to train with.

3.2.2 Data quality

In our particular case, we had two useful tools at our disposal: The provided "simple agent" and "random agent" agent types. When generating training data, one needs to learn both very good moves, as well as very bad moves. Because simple agents play the game at a human level, using games played between them seemed like a perfect opportunity to observe high-quality moves without learning them painstakingly through trial-and-error.

At the same time, relying only on simple agents to learn moves leaves one vulnerable to ignorance of the value of moves simple agents are not designed to make - both missed opportunities as well as obvious blunders. Q-Learning can only work if all choices in the Q Table are accurately updated to reflect their value. Otherwise, what might happen is that values remain set to their initialized value (0, in our case), which may be far higher than their actual value (e.g. a self-destruction a simple agent would avoid). See also chapter 2.1).

To reconcile these difficulties (i.e. to master the exploration/exploitation tradeoff), we created a new agent type which executes the move of the simple agent most of the time, and a random action the rest of the time. This "epsilon simple agent" was then used to generate bulk data for training. Initial results were promising: Agents easily solved the crate/coin tasks and engaged in aggressive behaviors when encountering an enemy. However, this was not the training regimen used in our final submission. For a more detailed analysis of agent performance, see chapter 4.2.2.

3.2.3 Limitations of learning from stored games

It is worth noting that a multiplayer game of this type is in principle non-deterministic. It is not possible to always predict what an enemy will do. Therefore, any single player policy used to train the Q Table introduces a bias into training - one learns to expect enemies to play according to the strengths and weaknesses of the policy that created the training data. The team was not familiar enough with the literature on game theory to derive a training strategy guaranteed to asymptotically converge to the "optimal" policy under the condition that enemy policies are unknown. The assumption in choosing the simple agent as a model, however, was that it would, at least in the beginning, be "good enough" to train a reasonably smart and capable agent.

4 Experimental results, observations and difficulties - (J. Weichselbaumer)

The following discussion will focus on the offline training approach using stored data to update a Q-table. For most of this project's lifetime, it was assumed that this was a viable way forward and would form the basis for our final submission. Due to complications that arose in the course of developing a framework that would allow our Q-tables to generate their own training data (described in 4.2) and the resultant time shortages, we eventually abandoned this approach and opted instead for a simpler implementation that used the rewarding framework provided by the game's code (see 2.5.4).

However, because this approach generated useful insights and data as well as working solutions and was conceptionally very similar to the training program with which we developed our final submission, we chose to use its outputs to create the graphs and statistics for this chapter.

4.1 Training time

Training our agents generally takes place in two stages: First, one creates training data, and secondly, one performs Q-table updates for every observation made in saved games.

Both the time taken to save a game and the time taken to train from a stored file depend on the step count of the game and the length of the Q-table; on a standard-issue laptop, training a long (> 250 steps) game for a short Q-table takes about 3 seconds, which can grow to around 20 seconds as the Q-table reaches sizes approaching 8e4. Playing and saving a game usually takes about 3 seconds.

The length of the Q-table in relation to the number of game steps trained with grows in an approximately square-root fashion, eventually converging to the maximum (finite) number of different states encounterable. During training, it became evident that the size of the vision field made a massive difference in the convergence of the Q-table. **??** shows that a vision radius of 3 produced Q-table growth at a pace roughly 80 times greater than a vision radius 1.

This makes sense from a theoretical perspective: If all vision squares have approximately similar (and independent) variance, this means that if there are n different in-game combinations for a $3 \cdot 3 = 9$ vision field, then for a $7 \cdot 7 = 49$ vision field there are $n^{\frac{49}{9}} \approx n^5$ times as many vision field combinations. If we assume (conservatively) n to be on the order of mag-

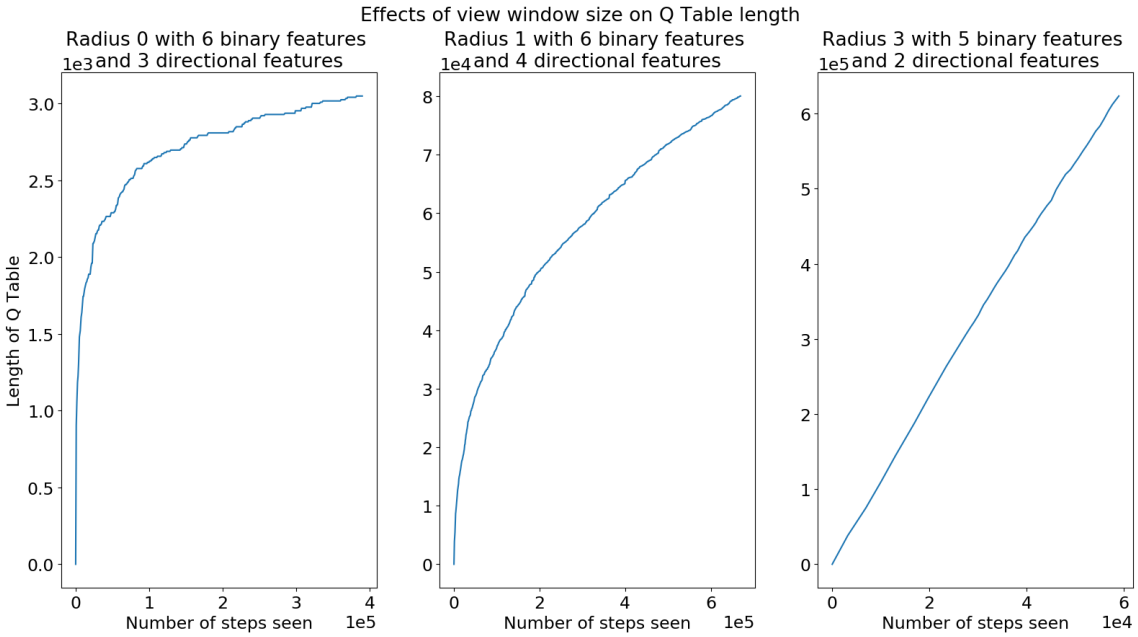

Figure 5: Using a vision field massively impacted Q-table growth

nitude of about 1e2, then this gives us about 1e10, or multiple billions of combinations. Making matters worse, as explained in the following paragraph, is that the implementation of our Q-tables was already rather inefficient to begin with, which meant that training a 7x7 model very rapidly slowed to a crawl and had to be abandoned.

During the course of development of the project, we debated whether we want to store the observation database in a sorted fashion, to make access more efficient (logarithmic time). Our implementation stores tables as numpy arrays, which are not dynamic and can therefore only be expanded by copying. We realized that, with linear growth in the number of new observations, the effort to maintain an expanding Q-table would grow quadratically. In addition, finding an observation in the table took linear time, due to our use of np.where to locate the observation vector in the database.

An initial effort was made to store the Q-table in a sorted way and to use a dynamic data structure in working memory to simplify the addition of new observations, but we ran into two difficulties: We did not find any "dynamic" numpy array-like data structures, as we had hoped to use, and did not want to give up the convenience and efficiency of only using numpy objects and methods. In addition, to our surprise, numpy did not seem to support "sorting" multidimensional arrays (in our case: Sort a matrix by its rows - first order rows by their first column's value, then order any equivalent rows by their second column's value, etc.), meaning we would have to have implemented this, ourselves, which seemed like a risky and error-prone approach.

Because run profiling in the initial course of our experimentations revealed that numpy functions of all kinds only took up a small fraction of total runtime, we came to the conclusion that our time would be better spent optimizing features and developing evaluation/training scripts rather than trying to improve the already acceptable speed of our implementation.

This was initially a reasonable solution, but this was only due to the fact that we did not intend on using vision fields with a radius larger than 2, anyway. Because our working solutions always used radius 1, our Q-tables always converged to a size less than 1e5, which meant that we were able to train models "overnight" and didn't have to worry about time taken.

Compounding this was the psychological effect of finally seeing an agent do reasonable things: Because agents always started behaving in clever ways at about 5e4 steps trained with, which was achievable in about 45 minutes and at which point updating the Q-table was still rather fast, we were too thrilled with the quality of our solution (compared to the failures and mistakes that plagued our initial implementations) to think a step ahead and question how performance would be impacted at another order of magnitude of training data. In hindsight, we should have tried harder to make accessing the Q-table more efficient; because Q-table length convergence sets in relatively quickly for small radii, expanding the table is not the main source of trouble. But because np.where searches the entire table even though we only seek 8 matches, this means that if table growth is approximately $\Theta(n^{\frac{1}{2}})$, the effort of merely looking up existing entries over the course of training grows in $\Theta(n^{\frac{3}{2}})$.

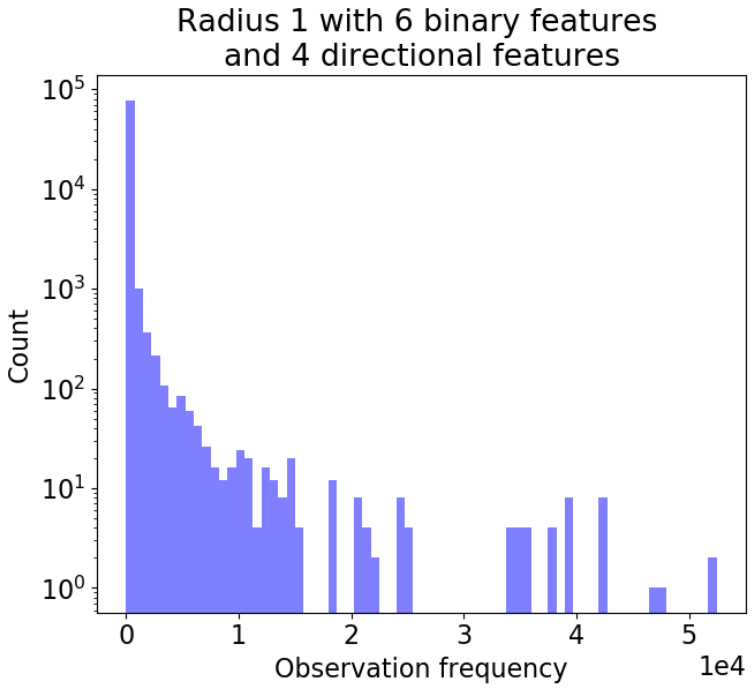

Figure 6: A minority of observations are seen extremely often, and the majority very rarely

Because most states are seldomly visited and a small number of states are visited very often, table index caching would have been a great speed boost and would not have required sophisticated changes to our implementation. We could probably also have hashed all indices to grant constant-time access, but, because of the psychological errors described above, we didn't fully consider the importance of doing this early on and were too busy, by the end, to implement it.

4.2 Agent quality assessment

4.2.1 Approach and implementation

The team was interested in deriving something like an objective benchmark for agent performance. The definition of a benchmark demands that any system of evaluations used is objective and repeatable. Playing an agent against itself is flawed for that reason.

An alternative might be to place the agent on a board with no enemies and time how long it takes for all coins to be collected, but this would risk training an agent to a good coin-collector, without regard for its ability to hold its own against live enemies.

The approach we settled on was to evaluate the quality of an agent by playing it against 3 simple agents a set number of games, counting the occurrence of each event, and then deriving the sum reward (as used in training) for that agent, for each game. The agent's score is the median reward earned over all trial games. Evaluating in this way has the direct advantage that it tests for ability to maximize reward against test data that are very similar to training data. It should, in theory, demonstrate the gradual improvement of the agent on test data through training and demonstrate the asymptotic maximization of reward by the Q-Learning algorithm.

To implement this evaluation framework, we wrote a script to alternatingly train from a set number of games and then run a suite of evaluations, noting the age of the model at the time of evaluation, the model's survival time against the simple agents, and the occurrence of each event for our model in each game.

We had written this script to support generating new training data through self-play so as to enable the agent to learn from the decision it would make in a game and their results. However, we encountered a mysterious and currently unexplained error which resulted in drastically slowed training speeds when alternating between training and evaluation, while leaving training speeds on the same data unaffected when calling the same function from a different script.

This was disappointing and forced us to abandon large-scale training through self-play, but the framework was still useful in automating and systematizing the process of training and evaluating and agent from stored games. Because agents only rarely witnessed the execution of those moves suggested by their own Q-tables, training very slowly "unlearned" any suboptimal moves suggested by the Q-table. The lack of self-play resulted in unwanted behaviors (such as bombs placed immediately after starting the game) only very slowly disappearing, which led to us abandoning the "from data" approach entirely, in favor of online learning through self-play.

4.2.2 Performance of Q-Learning and regression/classification

When planning out this project initially, the team assumed that any model chosen would frequently encounter unknown states and would therefore benefit from being augmented with a regression or classification model trained from Q(observation, action) values, in order to make informed decisions in novel states.

What this assumption oversaw was that a Q-learning model that encounters unknown states frequently is most likely under-trained and would therefore need to be replaced with a simpler one. In practice, the working solutions we used and refined almost never encountered unknown states, whereas the models that more often encountered them were not suitable as a final submission and were discarded early.

Therefore, regression models are mostly extraneous for the purpose of describing our agent's behavior. Nevertheless, because we devoted considerable planning to the task of selecting between regression models, it is worth briefly discussing their use for the task, here.

There were two main approaches we considered for this task: Either using kernel ridge regression to learn the (observation , action) -> Q(o, a) relationship and choosing the action that maximizes the value of the regression, or using a decision tree to map (observation) -> action, using the arg max of an observation's Q values as a training target.

The advantage of the former is that it is better equipped, in principle, to deal with sparse Q-tables. If only certain actions have been tried often under a certain observation (for instance, those actions an epsilon simple agent does most often), then the other actions will not have been updated sufficiently to function as training data for learning the Q function. Regression gets around this by simply using only those actions which have been tried often enough to have a reliably correct Q value and be useful as training data. A classification algorithm would falter here, because one could not trust that it would know the correct arg max, since most of an observation's Q values have not converged, yet.

We checked for these constraints using a running record of update counts for our Q-table. When applying regression, we used all (observation, action) tuples above a certain threshold of updates, along with their associated Q(o, a) values, as training data. For applying classification, we demanded that all actions in an observation's Q-table entry had been tried a minimum number of times before adding it to the training set.

In practice, classification of the table's arg max significantly outperformed regression. It should be noted, though, that the team did not perform a very sophisticated analysis to discover the right hyperparameters of the kernel regression. We used scikit learn's implementation of kernel ridge regression (with default parameters and a polynomial kernel) and found the results disappointing.

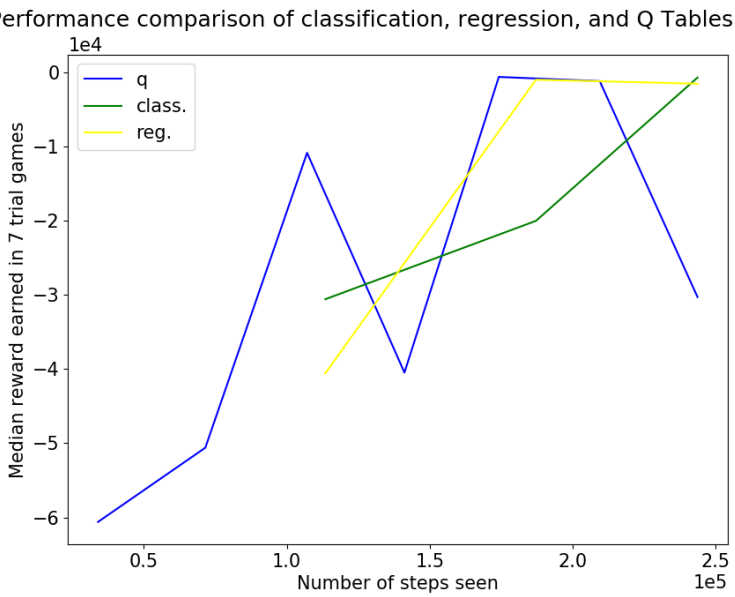

Figure 7: Intermittent evaluations during training were used to compare the performance of Q functions against their learned approximations.

Forcing the agent to make decisions using only regression/classification results in moderately and significantly worse performance, respectively. The above figure captures the asymptotically increasing behavior of evaluation performance, but hides the fact that regressions and classification often resulted in "deadlock" situations in which an agent stayed in one spot, waiting, or simply moving between two fields until disturbed by another player; this was particularly frequent in the case of regression.

In practice, regression/classification agents only made intelligent decisions in the first few steps, if at all, after which they only waited or strayed about. While the overall amount of "reward" earned improved, this does not necessarily correspond to improved performance in-game; the bulk of negative rewards come from invalid actions, which the agent learned to avoid. While the agent might have ended up avoiding punishment better than before, regression/classification methods by no means resulted in the kind of behavior (coin-collecting, enemy-destroying) that would score points under tournament conditions.

Keep in mind that these example data come from a well-trained radius = 0 agent with 10 features. Performance of regression/classification would probably be comparatively dreadful if we had trained from a much larger and very sparse Q-table - a typical scenario with large vision radii.

Q-Learning, on the other hand, resulted in consistent improvements in observable behavior. Agents learned to engage in useful actions and avoid death. As described in chapter 2.4, how quickly an agent was able to learn the utility of an action depended also on how

well-connected the action and its reward was; the value of bomb placements took significantly longer to learn than the (negative) valid of executing an invalid move. Because agents preferred not to sit and wait, they also thereby exposed themselves to more risk, which explains the at times lower rewards collected during the trials, compared to regression/classification.

5 Outlook - (J. Weichselbaumer)

5.1 Retrospective evaluation of our approach

Carrying out this project revealed that Q-Learning is not in principle a bad choice for playing Bomberman; agents eventually learned to exhibit expected in-game behavior such as avoiding death and invalid actions, destroying crates, collecting coins, and attacking enemies. However, this assessment comes with some qualifications:

5.1.1 Dimensionality

As was shown in chapter 4.1, the dimensionality of the Bomberman problem spirals out of control even when looking at less than a quarter of the playing field at one time. Any implementation of reinforcement learning that involves maintaining a database of observations is completely inadequate for the task of using the whole (or even a large portion of) the board state to make decisions. Even when disregarding the excessive amount of time spent training the model, the space in memory needed to hold the Q-table and observation database for a close-to-completely trained 7x7 agent would exceed 1 GB by at least an order of magnitude.

5.2 Feature selection

Because of the above constraints, it became necessary to practice dimensionality reduction by looking at only a small subset of the field and enriching this view with hand-crafted features (see chapter 2.3.1).

Here, we get to the crux of the problem with Q-Learning for this task: The only way to keep dimensionality under control is to introduce handmade features, but the existence of those features presupposes that a domain expert is present and able to refine the complex feature space of Bomberman into a handful of features that correlate extremely well with in-game rewards.

In other words, the only way to solve Bomberman with Q-Learning is to already know all the solutions. Training a bot to play Bomberman with the help of a few extremely compact and useful features is "nothing special"; the agent never actually develops any interesting internal representations of the game the way, for example, a deep neural network would.

The problem is only tractable because there is a human there to provide strong cues for how to behave in-game.

An immediate example of this is the "dead end" problem: There are situations where placing a bomb and then moving in a certain direction guarantees (barring enemy interference) death.

What makes this problem tricky is that it is not possible to always avoid these situations unless the agent can see whether a path away from a bomb forms a dead end. Because bombs explode with radius 3, one would need to be able to see at least 4 coordinates in every direction to be able to verify whether a dead end situation is or is not present. To implement this, one would need to find a clever way to allow for this knowledge (e.g. by eliminating other parts of the view field), because simply setting the view radius to 4 would have unfeasible dimensionality problems, as explained above.

The solution we came up with is to check whether such a situation is present and indicate it with a boolean. As expected, the agent learned to follow our "direction to safe field" feature in those situations. But again, this approach is not a very inspiring solution, and wouldn't be an option in high-dimensional problems that the human administrator does not already understand.

Were one to omit these features and simply rely on a large window and expect the agent to develop its own statistical intuition for how to behave, one would need an extreme volume of training data, corresponding to the number of different states (see chapter 4.1). Furthermore, many choices made only have consequences after a few steps, e.g. laying a bomb now will only in 4 moves cause an explosion and deliver a reward. To learn these temporally displaced associations, one needs to repeatedly carry out the actions in order for the reward to propagate backwards through the Q-table and create the association between dropping a bomb now and being rewarded for it later, further increasing the training volume necessary.

One can get around this problem (as discussed in 2.4) by immediately rewarding the agent when it follows a certain useful feature in an appropriate situation. This ensures that agents learn useful behaviors quickly, but it also means that the entire process of using Q-Learning is redundant, because we not only define the features, but also their value. To be able to do this means the Bomberman problem is already solved and the application of Q-Learning is reduced to something like a recreational exercise.

One could humorously speculate that one of the best features we could have constructed is simply an integer value in $[0;5]$ that encodes the decision a simple agent would make in the exact same situation - while it would probably not have been accepted as a solution, it presents a fair elucidation of the whole feature selection problem. If Q-Learning cannot be efficiently used to find new insights, why use it at all?

5.3 Recommendations for improving the lecture

Because the other sections have already explained some of the problems we encountered and the lessons we learned through hindsight, this section will focus mostly on the suitability of the task itself.

First of all, the team would like to thank the directors of the lecture for providing such an in-depth, rigorous, and challenging introduction to the field of machine learning. The idea of using a project of this sort is excellent; it trains teamwork, engineering skills, planning ability, ability to compare and contrast different approaches, and demands the ability to independently evaluate and understand what an algorithm actually does and how it is used, rather than memorizing this or that formula in order to repeat it on an exam.

Bomberman itself is not a perfect choice for a game. As explained, one is forced, if using a reinforcement learning approach of the kind we used, to practice feature reduction to the extent that creating the agent essentially amounts to writing a bot and waiting for the table to learn its behavior. This kind of approach does not capture the beauty and potential machine learning is known for: Uncovering hidden structures and meaning in complex data.

If one doesn't want to use features of that sort, one has to train a model for a very long time, which can cause a crisis if a minor mistake is discovered at the last second, and the last 48 hours of training were wasted. Any solution demanded by a machine learning project should, in principle, not be primarily limited by time spent training. A good project would require ingenuity and cleverness, but not powerful servers and brute time training, as this would create an uneven competition right out of the gates between those with access to powerful computational resources and those without.

A tentative suggestion for an alternative game would be Super Smash Brothers: Melee (SSBM). The reasoning here is that the feature space is, on the face of it, simpler. Combat between players is relatively simple; one has access to few actions, and the map is constant and unchanging. Players are easily described by their character, health and x, y coordinates. One could disable certain power-ups which might complicate gameplay unhelpfully. The fact that SSBM enjoys an avid competitive following suggests that there is sufficient depth and complexity to the game to make it an interesting task to solve.

A choice such as this would have the advantage that students would be less incentivized to hand-craft features (as gameplay is less easily described using heuristics, and it's not very straightforward to determine the "next action", unlike in the game Bomberman, in which one can easily incentivize the agent to move to a particular square and drop a bomb there), while neither trivializing the feature space nor exposing it so such enormous dimensionality problems.

The downside to this choice is that SSBM is a proprietary, most likely closed-source game and almost certainly doesn't expose APIs for easy access to game events. It might be legally

"sketchy" or difficult to implement this, though the author hasn't fully researched the matter.

Bibliography

Richard S. Sutton and Andrew Barto. *Reinforcement learning*. A Bradford book. MIT Press, Cambridge, Mass. [u.a.], 3. print. edition, 2000. ISBN 0-262-19398-1 and 978-0-262-19398-6.

Csaba Szespari. *Algorithms for reinforcement learning*. Number ARRAY(0x563223f2ba78) in Synthesis lectures on artificial intelligence and machine learning. Morgan Claypool, San Rafael, Calif., 2010. ISBN 978-1-608-45492-1. doi: 10.2200/S00268ED1V01Y201005AIM009. URL http://www.morganclaypool.com/doi/abs/10.2200/S00268ED1V01Y201005AIM009. Bibliographie : p. 73-88.