# Training a Bomberman Agent using Reinforcement Learning [1]

Phillip Kollenz and Bernd Mumme

[1] Fundamentals of Machine Learning Final Project, WS 2020/21

# Contents

# 1 Introduction

In this report, we will explain how we trained an agent to play classic bomberman using reinforcement learning techniques. The agent will play against three other agents, where each agent starts in one of the four corners of the playing board, see Figure 1. BM

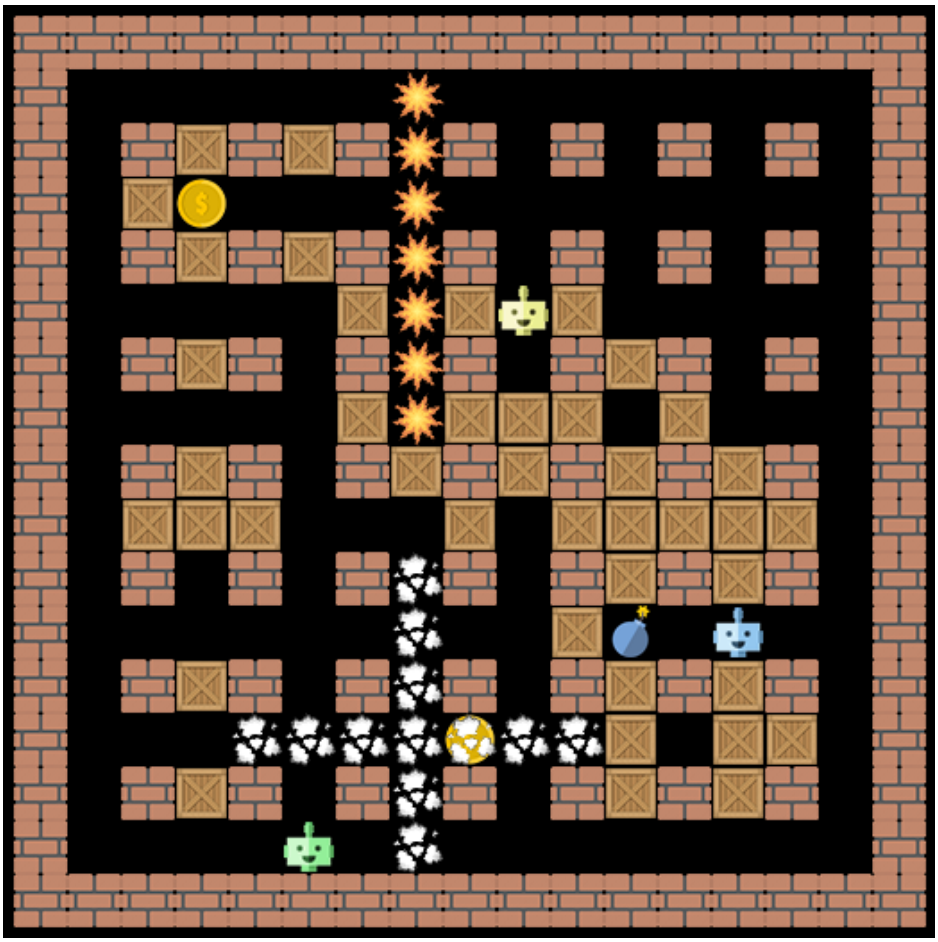


Figure 1: Bomberman board scheme

The agents have to destroy crates, collect coins, and destroy the other agents to be the last ones standing and collect most points. Collecting a coin gives one point, destroying another agent gives five points.

The agent trained by us will have to learn how to navigate this playing field and hold its own in a set of three tasks:

1. On a game board without any crates or opponents, collect a number of revealed coins as quickly as possible. This task does not require dropping any bombs. The agent should learn how to navigate the board efficiently.

2. On a game board with randomly placed crates yet without opponents, find all hidden coins and collect them within the step limit. The agent must drop bombs to destroy the crates. It should learn how to use bombs without killing itself, while not forgetting efficient navigation.

3. On a game board with crates, hold your own against one or more opposing agents and fight for the highest score.

We will be training a model with a traditional Reinforcement Learning (RL) approach in a decision tree, described in subsubsection 2.3.2.

Each of the models will be explained and its performance on the given tasks evaluated, specifically with the mindset of which of our agents should compete in the tournament.

We will also explain our feature and reward selection in detail.

## 1.1 Setup

Table 1: Packages used in this project.

| Packages |
|---|
| Pickle |
| SciPy |
| SKlearn |
| H5 |
| numpy |
| matplotlib |
| pandas |
| PyGame |
| tqdm |

We divided the report into two parts, so that each half is written by one of us. At the beginning of each half, a small initial on the righthandside will indicate that the author has switched (BM), (PK).

The `git`-repo with our code can be found under `https://github.com/phil1425/bomberman_private`.

# 2 Theoretical Background

This section will give a theoretical overview over all the Machine Learning (ML) techniques used in this paper. We explain the regression techniques used to decide which features are most important to train on, curiosity-based decision making, which we use for the rewards, as well as decision trees and actor-critic learning, our two main training techniques for the final agents.

## 2.1 Techniques for Feature Selection

While we can design all the features we want, we somehow have to decide how important each one is for the agent to learn well and converge as quickly as possible. This can be done by testing the features against a validation set of game states played by the rule-based agent.

This section will explain the three algorithms we use to score our features. We chose one filter-based method, the Pearson correlation (subsubsection 2.1.1), one wrapper-based method, recursive feature elimination (subsubsection 2.1.3) and one embedded method, a random forest as described in subsubsection 2.1.4, to select our features.

Filter-based selection methods measure the relevance of each feature and select features on a given threshold. They are relatively robust against overfitting [5].
Wrapper-based feature selection methods select features through evaluation on subsets of the variables, which means it discover relations between the variables and select the most useful ones. They are, however, dependent on the classifier.
Embedded feature selection methods function similarly to wrapper-based methods, but they use an intrinsic model-building metric, which makes them less computationally expensive and more robust than wrapper-based methods [5].

### 2.1.1 Pearson Correlation

The pearson correlation evaluates whether there is evidence for a linear relationship between pairs of variables in a dataset. The pearson correlation between two variables $x$ and $y$ is defined as

$$\rho_{x,y} = \frac{\text{cov}(x,y)}{\sqrt{\text{var}(x)} \cdot \sqrt{\text{var}(y)}} \tag{1}$$

where $\text{cov}(x,y)$ is the sample covariance between $x$ and $y$, and $\text{var}(x)$ and $\text{var}(y)$ are the variances of $x$ and $y$ respectively [6].

This correlation can take values in the range $[-1,1]$. The sign indicates the direction of the correlation, while the magnitude indicates the strength of the relationship. For a simple threshold classification like we are looking for here, only the magnitude of the correlation is important.

### 2.1.2 Logistic Regression

For the next two models we will be using logistic regression. Logistic regression tries to find parameters $\beta_0, \beta_1$ for the sigmoid function

$$\sigma_x = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x)}} \tag{2}$$

by Maximum Likelihood Estimation (MLE) [1].

### 2.1.3 Recursive Feature Elimination

Recursive feature elimination (RFE) [4] is an implementation of the backward feature elimination. It eliminates single or multiple features recursively until a given threshold is reached. The algorithm works as follows:

1. Train the classifier (here logistic regression).
2. Compute the ranking criterion for all features.
3. Remove the feature with smallest ranking criterion.

Note that the features this method selects might not be the features with most singular importance, but instead the selected subset of features is of importance.

### 2.1.4 LASSO Regularisation

LASSO (Least Absolute Shrinkage and Selection Operator) regularisation is a way to force weights in the logistic regression to be zero, such that the model already performs the regression on an optimised subset of the data.

$$\hat{\beta} = \text{argmin}_{\beta} ||\mathbf{y} - \mathbf{X}\beta||_2^2 + \tau||\beta||_1 \tag{3}$$

The estimator is calculated via the L1−norm.

## 2.2 Policies

This section discusses two policies for action selection and lists their advantages and disadvantages.

### 2.2.1 Epsilon-greedy Algorithm

In ML scenarios like the one at hand, there is always a trade-off between two distinctive action paths: Exploration and Exploitation [3].

Exploration allows the agent to learn more about each action, to improve its estimates of the values of actions, and enables it to make more informed decisions in the future.

Exploitation chooses the action that gives the most reward based on the agents current estimates. Since the estimates the agent has at any give time are probably not the best estimates it could make, there is a danger of the agent exploiting its current actions to get the most reward it knows how to, instead of trying new actions that might give less reward initially, but lead to the agent learning more about its environment and the actions it can perform, and therefore a higher reward later-on.

Because the agent cannot exploit and explore at the same time, we use an action selection mechanism called epsilon-greedy method. It simply selects the greedy strategy most of the time, which a small chance $p < \varepsilon$ of exploring:

$$a(t) = \begin{cases} \max\left(Q_t(a)\right) & \text{with} \quad p = 1 - \varepsilon \\ \text{random } a & \text{with} \quad p = \varepsilon \end{cases}, \tag{4}$$

where $a(t)$ is the action taken at time $t$, $Q_t(a)$ is the reward function of such an action and $\varepsilon$ is a small parameter. This parameter $\varepsilon$ can be slowly decayed in training to allow for larger learning potential at the beginning and slower learning but more greedy play when it is obvious that the agent already knows what it is doing.

### 2.2.2 Softmax

The epsilon-greedy policy chooses an explorative action at random. In scenarios where bad actions are particularly bad, this might be undesirable [7]. This is where we might want to use a softmax policy. This policy chooses actions according to a probability distribution of the form of a sigmoid function as shown in Equation 5.

$$S(x) = \frac{1}{1 + e^{-x}} \tag{5}$$

This ensures that the agent still chooses its preferred features most often, but has probabilities according to its current ranking of an actions importance to pick them as well.

## 2.3 Models

In this section we will describe different models we will use to teach our agent to play.

### 2.3.1 $Q$-Learning

As a base-model, we introduce the $Q$-learning algorithm [7]. It is an off-policy algorithm, which means it needs an external policy, two of which we discussed in subsection 2.2.

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
  Initialize $s$
  Repeat (for each step of episode):
    Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Take action $a$, observe $r$, $s'$
    $Q(s, a) \leftarrow Q(s, a) + \alpha\big[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\big]$
    $s \leftarrow s'$;
  until $s$ is terminal

Figure 2: $Q$-learning algorithm: $Q(s, a)$ is the estimated reward from a state and an action and $r$ is a reward function[].

The algorithm works as follows: It initialises the $Q$-function, chooses an action through an external policy as described above, takes the action, observes the new state and gained reward, and updates $Q$ accordingly. Because $Q$ converges to the optimal $Q^*$, we now only have to repeat this process until it converges.

### 2.3.2 Density and Decision Trees

Decision trees can be visualised as classifying a pattern through a series of questions that can be asked in a "yes/no" style. The tree starts with a root node, then splits off into branches until each

branch ends in a leaf, which have no further connections. This means it partitions feature space into arbitrary decisions that are based on the Gini impurity, which for a node $l$ is defined as

$$\mathrm{Gini}_l = N_l \left(1 - \sum_{k=1}^{C} \frac{N_{lk}^2}{N_l^2}\right), \tag{6}$$

where $N_l$ is the total number of instances in node $l$, and $N_{lk}$ are the number of instances of class $k$ in $l$. One big advantage of decision trees is its ability to visualise decisions easily.

For a regression instead of a classification problem, we can instead use a density tree, for which the perfect splits minimise the leave-one-out error of the resulting children:

$$\mathrm{looErr}_l = \frac{N_l}{NV_l}\left(\frac{N}{N_l} - 2\frac{N_l - 1}{N - 1}\right), \tag{7}$$

where $N_l$ and $V_l$ are the number of instances in node $l$ and its volume, and $N$ is the total number of instances for the class under consideration.

### 2.3.3 Random Forest

A random forest consists of multiple decision trees [2]. The random forest prediction is the average of all simple tree predictions. This leads to a more sturdy prediction at the cost of a higher computation time.

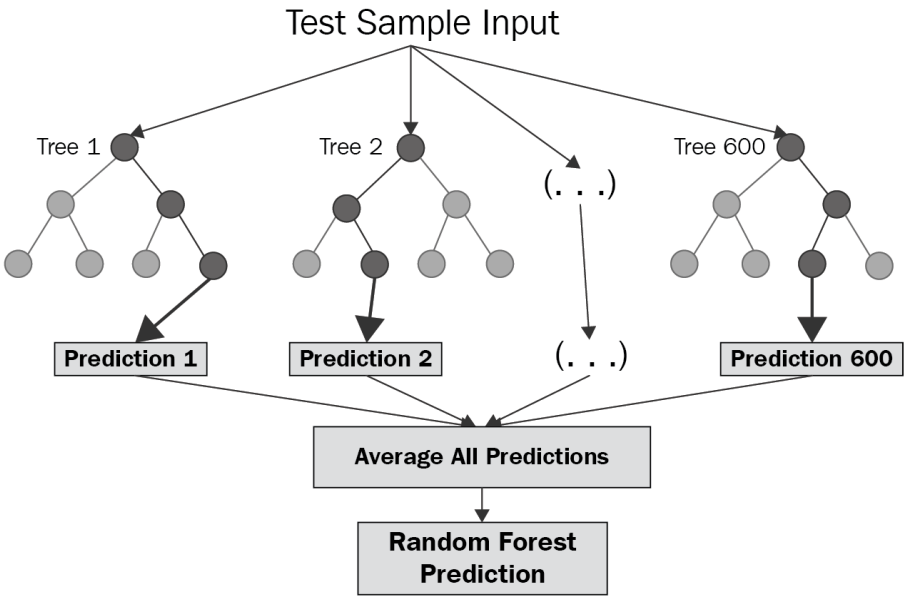



Figure 3: Random Forest estimation. Instead of just evaluating one tree, the Random Forest evaluates many trees and averages the predictions [2].

# 3 Computational Process

## 3.1 Feature Design

### 3.1.1 Handmade Features

We handdesign features because computing them via Deep Learning techniques would be too computationally expensive for the purpose of this paper. To test the importance of these features, we run a suit of three feature selection algorithms (as described in subsection 2.1) over a set of roughly 45000 games of the rule-based agent and extract the most important features. This way we have a starting point and know which features could be good for our Reinforcement Agent to start learning quickly. A short analysis of this "feature benchmark" will be given at the end of this section.

In total we designed 34 features that we deemed important. In the following they will be individually listed and explained.

**Step number**

A simple counter of the current step number, for the agent to correlate actions to specific points in the game and change its strategy accordingly.

**Sum of crates in radius 1/2 around the agent**

These are two features that add up all crates around the agent either in a radius of one tile or in a radius of two tiles. This is an important measure of enclosedness for the agent.

**Crates that explode if the agent sets a bomb at the current position**

Counts the crates that would be in the explosion radius of a bomb if it is placed at the current location. This gives the agent a measure for the possible reward in its immediate circumference.

**Agents that explode if the agent sets a bomb at the current position**

Counts the agents that would be in the explosion radius of a bomb if it is placed at the current position and time. Does not account for the fact that agents can run out of this explosion zone, that would be too advanced of a feature.

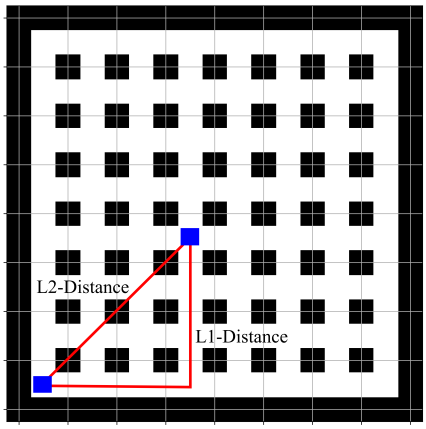
**Distance to next agent**

Calculates the L1-distance

$$||\mathbf{x} - \mathbf{y}||_1 = |x_2 - x_1| + |y_2 - y_1| \tag{8}$$

between the agent and its next opponent. This gives the agent a measure for the danger it is currently in.

We use the L1-distance, because on a playing board like the one in Bomberman, where walls permanently force the agent to move in straight lines, it is the fastest way to a specified target, see below. Here, the L2-distance is the standard euclidian norm as we know it. The L1-distance, on this kind of board, on the other hand, is equal for all paths between the one that is shown in the figure and every other direct path between the two blue points, up to the mirrored path that goes three spaces up and then three spaces right.




**Relative coordinates of the next agent**

Calculates the relative coordinates of the next agent. This tells our agent which direction it has to go to either escape or chase the opponent. Gives back two features, $x$ and $y$.

**Crates in radius 1 around the next agent**

Counts the crates in a radius of one around the next opponent. This gives the agent a bit of far-sight on the board position as well as a measure for how enclosed the next agent is.

**Agent's own position**

Just gives the agent its own position as two features, $x$ and $y$.

**Coins in radius 3 around the agent**

Counts the coins in a radius of three tiles around the agent. This provides the agent with a general sense for the reward in the area.

**Distance to next coin**

Calculates the L1-distance (see Equation 8) between the agent and the next coin. Again, this gives the agent a sense of how close the next big reward is.

**Relative coordinates of the next coin**

For the agent to also know where to go to collect this reward, we also provide it with relative $x$ and $y$ coordinates of the next coin.

**Crates in radius 1 around next coin**

To get an additional measure of how enclosed the next coin is (even though this might be somewhat predictable from a combination of the relative variables and the L1-distance), we provide the agent with the sum of crates in a radius of one around the next coin.

### Is bomb setting possible?

Gives the agent a boolean for whether or not bomb setting is possible. This is important to know since it almost always correlates with both the immediate danger to the agent by the bomb he just set, if it is not able to set a bomb, and the possible reward to gain if it is able to set a bomb.

### Agent's scores

Gives the agent four features with its own score and the other three agents scores in order. This might provide it with a sense of how urgent it is to play a certain way to get the most points, or keep the most points.

### Sum of all crates on the field

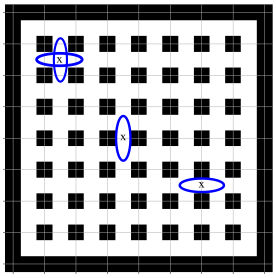
Counts all the crates on the playing field and returns the sum. This might be a measure for what general strategy of play to prioritise: Collecting coins, destroying crates, or destroying opponents.

### Is the agent at one of the borders of the playing field?

Returns two features that tell whether the agent is either at the top or bottom border, or if it is at the left or right border.

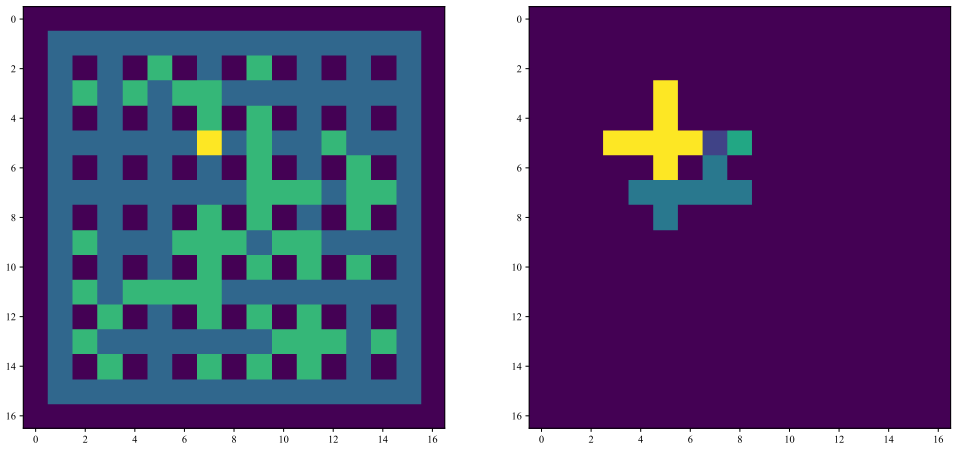
### Agent's relative position to the inner walls

This feature tells the agent whether it is at an intersection, a place where there are walls to the left and right or at a place where there are walls to the top and bottom, as seen in the figure to the right. This should be an important feature to know because it eliminates some movement options and gives a sense of enclosedness.



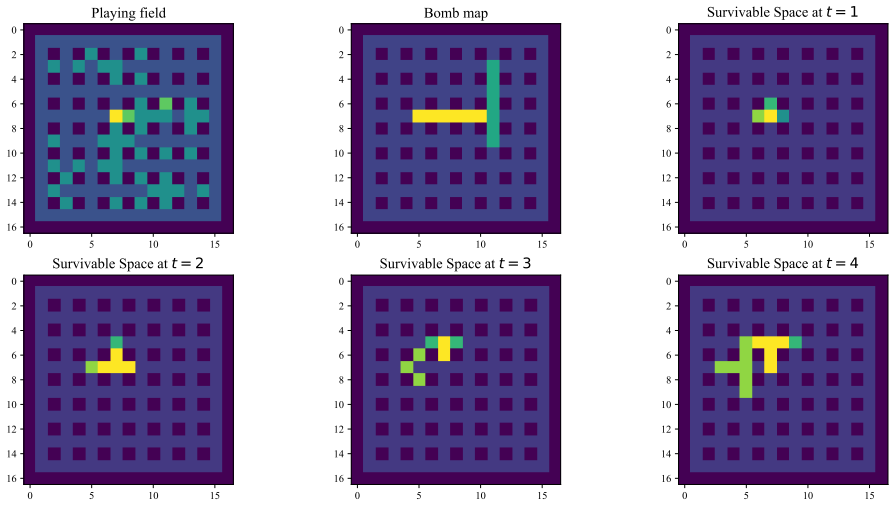
### Free space in all four directions

This feature for spaces free of crates to go on in each of the four directions using a tree search algorithm, and returns each of them as a feature. It scans the free space in one direction up to a defined maximum of free spaces, and with that gains foresight as to what routes it can take and how they are connected, and where there are crates. These four features are, together with the five next ones, the most advanced features in our set. The yellow dot in the left figure symbolises the agent, the bright green spaces are crates. The right figure visualises what the agent sees as free spaces in each direction.



### Survivable space for all five actions

This feature adapts the tree-search devised for the features described above slightly. It aims to, instead of showing free space in each direction, give back the number of squares on which the agent can end up and survive after a given time period, and instead of in each direction, calculate this for all movement actions, `UP`, `DOWN`, `RIGHT`, `LEFT` and `WAIT`. It achieves this by comparing the bomb map at each time step with its possible actions.



We see the board in the top left corner, this time indicating bombs with a green marker, then in the next plot to the right, the bombs and their respective cooldowns. The next plots show us, in order, which spaces the feature recognises as safe to land on after the specified number of timesteps, and for each action tried.

This method leads to an agent that is able to see all possible futures of where it will survive and how at once [8]. It can therefore learn how to behave when bombs are placed in certain situations where there is not much space to escape, or how to react to overlapping bombs.

### 3.1.2 Benchmarking Features against the rule-based Agent

To test the goodness of our self-designed features, we tested them against a set of games played by the rule-based agent and used three different feature-selection methods to score how important several features are to the rule-based agent.

Figure 4 shows the actual correlation strength of the Pearson correlation. We can clearly see that 8 features are correlated to the next action the agent takes way more strongly than all others. Unsurprisingly, these features are 8 of the 9 tree-search-based features, which give the deepest insight into the agents surroundings, only missing the feature for the `WAIT` action in the survival tree, since the rule-based agent almost never waits on purpose, unless it has to.
The other feature scoring methods did not have as easy of a score to retrieve, so we will not be comparing them side-by-side.

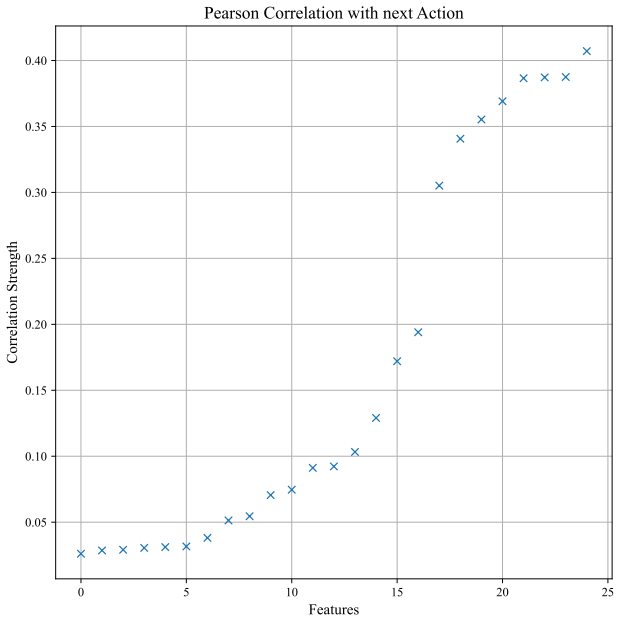



Figure 4: Pearson Correlation strength plotted for each feature.

Table 3: A benchmark for the importance of features. Each of our 34 features is evaluated with each of the three feature selection methods, and then they are ranked by the sum of the feature selection methods that selected any particular feature. This table shows the top 25 features.

| Feature | Pearson | RFE | Lasso | Sum |
|---|---|---|---|---|
| Survivable spaces up | True | True | True | 3 |
| Survivable spaces right | True | True | True | 3 |
| Survivable spaces left | True | True | True | 3 |
| Survivable spaces down | True | True | True | 3 |
| Sum of crates radius 2 | True | True | True | 3 |
| Sum of crates radius 1 | True | True | True | 3 |
| Sum all crates | True | True | True | 3 |
| Next agent rel y | True | True | True | 3 |
| Free space up | True | True | True | 3 |
| Free space left | True | True | True | 3 |
| Free space down | True | True | True | 3 |
| Distance next agent | True | True | True | 3 |
| Crates explode if bomb here | True | True | True | 3 |
| Crates around next coin | True | True | True | 3 |
| Bomb possible | True | True | True | 3 |
| Agents explode if bomb here | True | True | True | 3 |
| Agent pos rel to wall | True | True | True | 3 |
| Survivable spaces wait | False | True | True | 2 |
| Step Number | False | True | True | 2 |
| Other Agent 2 score | False | True | True | 2 |
| Agent at top/bottom border | True | False | True | 2 |

### 3.1.3 Data Augmentation

A empty game board of Bomberman has symmetry in respect to mirroring parallel to its vertices PK and its diagonals, as well as 4-fold rotational symmetry. As a result, any valid game state transformed by these operations is also a valid game state. This can aid the training process in one of two ways:

From one game state, seven other game states can be generated by applying mirror- and rotation operations. If the corresponding action is also transformed in the same way, the augmented feature:label pairs can be added to the training data. This effectively increases the amount of available training data 8-fold.

The model can use features that are invariant to those transformations. Any feature that gets extracted from the game state can be modified in this way by transforming all incoming game states in such a way, that the position of the agent is always in the same of eight possible segments. This is done by locating the segment of the agent and then applying mirroring and rotation operations accordingly (Figure 5).

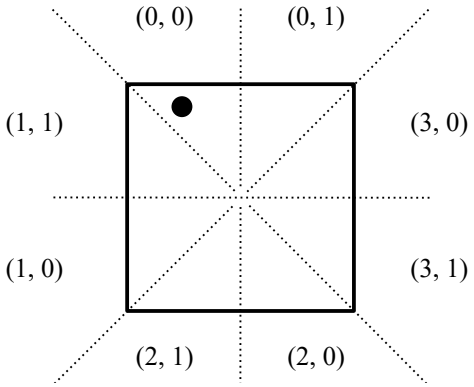



Figure 5: Lines of symmetry and required operations $(n_{\text{rotation}}, n_{\text{mirroring}})$ for each segment of the field to keep the agent in the upper left segment (black dot).

Since the game board is not continuous, but made of 17x17 squares, special attention has to be given to the edges between segments. While the decision is arbitrary, it needs to be consistent along all 8 segments. In Figure 6, the edge cases for all possible positions of the agent are shown.

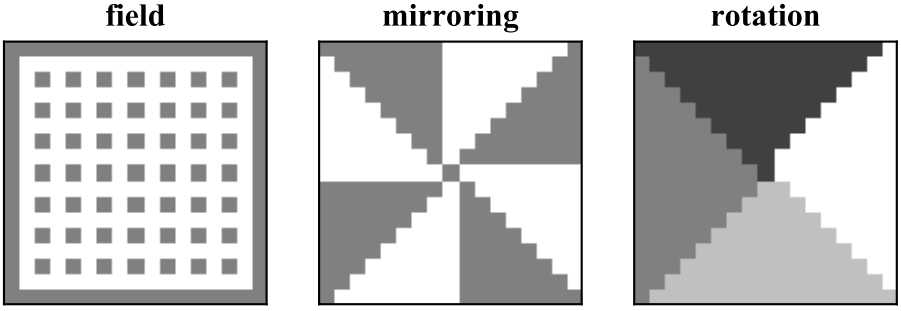



Figure 6: Possible positions of the agent and segments for mirroring and rotation with all edge cases

## 3.2 Reward Design

### 3.2.1 Handmade Rewards

The goal of Bomberman is to have the highest score after 400 steps. The score is increased by one if a coin is collected, and by 5 if an enemy agent is killed by a placed bomb. The coins are obstructed by crates, which have to be removed by placing bombs and waiting for them to explode. In order for an agent to obtain the smallest unit of reward, it has to be able to:

1. Place a bomb.
2. Find cover from the explosion.
3. Wait until the bomb is exploded.
4. If a coin is released, collect the coin.

This means that the auxiliary rewards have to be designed for the agent to be able to start learning. Ideally, the $Q$-function of the agent should approximate the final reward at the end of the round, since this is a good indicator for the odds of winning the game. But having the same reward for all steps taken during an episode leads to slow training or no progress at all: At the start of the training, the decisions are chosen almost entirely at random due to $\varepsilon$. If the agent now receives some reward, the q values for all actions taken will be increased. This includes invalid actions, waiting, moving in the wrong direction or actions taken after the reward is already collected. In order to speed up training, reward has to be attributed to actions that lead to the collection of the reward.

First, the rewards are divided into general rewards and immediate rewards. General rewards are received if the agent acts in a way that is beneficial to the overall score. The action that lead to this reward does not have to be the timestep where the reward was earned. Immediate rewards are received for actions that are obviously not beneficial to the training and the action and the resulting reward are in the same timestep.
The estimated reward for every action is then the sum of all future general rewards $r_t$, reduced by a discount factor $\gamma$ to incentivize short-term rewards over long-term rewards. The immediate reward $i_t$ is added for each step individually, since it is supposed to influence only a single timestep. The sum of both rewards can then be used as the target Q-value.

$$\begin{aligned} R_t &= r_t + \gamma \cdot R_{t+1} \\ Q_{\text{target}}(s_t, a_t) &= R_t + i_t \\ Q_{\text{new}}(s_t, a_t) &= Q_{\text{old}}(s_t, a_t) + \alpha(Q_{\text{target}}(s_t, a_t) - Q_{\text{old}}(s_t, a_t)) \end{aligned}$$

The BombeRLe environment offers a range of events from which auxiliary rewards can be designed. For each event, a general and immedeate reward are assigned (Table 4). Since the required incentives change during training, some of the rewards are dependent on the index of the current round $n$.

Table 4: Events for auxillary reward calculation

| event |
|---|
| `INVALID_ACTION` |
| `WAITED` |
| `CRATE_DESTROYED` |
| `COIN_COLLECTED` |
| `KILLED_OPPONENT` |
| `BOMB_EXPLODED` |
| `KILLED_SELF` |
| `GOT_KILLED` |
| `SURVIVED_ROUND` |
| Agent explores new square |
| Agent visits old square |

A problem that frequently arises is the formation of loops. Since returning to a already visited square is less likely to cause an `INVALID_ACTION` event than exploring a new square, the agent

often chooses to return to its previous position. This leads to a loop where the agent only moves back and forth. To counteract this, we add another auxiliary reward that incentivizes exploration: Every time the agent visits a square it has not entered before, it receives some additional reward. If the agent visits a square it has already seen, it receives a penalty.

## 3.3 Model Design

### 3.3.1 Decision Tree

The number of possible states for the game is too large to generate a Q-Table directly. A rough estimate for the possible number of states can be made by calculating the number of arrangements of crates on the field: There are 176 free spaces on the game board, resulting in $2^{176}$ possible arrangements of crates. Even without including any other variables of the game state, a $Q$-Table containing every possible state of the board is not possible. To overcome this problem, the number of possible states can be either reduced by selecting features from the game state as done in Section 3.1, or a more memory-efficient model can be applied. For the constraints of this project, a decision tree fits very well. The feature space of the game is discrete and only has a few possible values fore each feature. This reduces the number of possible thresholds for the decision tree to consider, which speeds up the building of the tree. While the training of a decision tree is comparatively slow, inference is very fast due to its hierarchical structure. This means that computation time requirements during the tournament can be easily met. At small scales, decision trees can be visualized to make their decision interpretable. This can aid in debugging the model.
Once built, a decision tree cannot be changed. This means that for each training step, the tree has to be rebuilt from the training data. Instead of storing only the experience from the last episode, as would be possible with gradient-based methods, training data from the last 1M steps are stored in a circular buffer. Afer every 20 episodes, the tree is rebuilt with the new data. Here, the predictions of the old model can be used to build the new one: For every action that was not taken, the predicted q value of the old model is used as training data for the new model.

**Algorithm 1:** Decision tree learning

```
Initialize buffer.
Initialize policy with ε = 1.
For n steps do:
    Collect experience from environment, add to buffer.
    Calculate Q_target from recorded events.
    Predict Q_old values for states in buffer.
    For every state in buffer:
        Q_new = Q_old for all actions not taken.
        Q_new() = Q_old + α(Q_target − Q_old) for the action taken.
    Build new tree from buffer.
    Change Policy to tree model with decaying ε.
```

## 3.4 Training

### 3.4.1 General Procedure

### 3.4.2 Coin collection on empty board

For training on an empty board with no opponents, only features related to movement or the collection of coins are needed (Table 6). Since the task does not require a complex movement pattern to obtain the first coin, the reward design is fairly simple (Table 5). Training was done using a decision tree model with a circular experience buffer with 10k steps and a minimum of 10 leaves per node. As a policy, $\varepsilon$-greedy was used with a starting value of $\varepsilon = 0.5$ which decays with a lifetime of $\tau = 200$ episodes to a final value of $\varepsilon = 0.05$.

Table 5: Assigned rewards for task 1

| event | | |
|---|---|---|
| `INVALID_ACTION` | $-0.05$ | $-0.1 - 1.9e^{-n/200}$ |
| `WAITED` | $-0.05$ | $-0.1 - 1.9e^{-n/200}$ |
| `COIN_COLLECTED` | 1 | 0 |
| `KILLED_SELF` | $-5$ | 0 |
| `SURVIVED_ROUND` | 2 | 0 |
| valid action | 0 | $e^{-n/200}$ |
| Agent visits new square | 0.05 | 0 |
| Agent visits old square | $-0.05$ | 0 |

Table 6: Selected features for task 1

| name of feature | number of features |
|---|---|
| distance to next coin | 1 |
| relative coordinates of next coin | 2 |
| obstruction level of next coin | 1 |
| agent position relative to wall | 1 |
| free space in each direction | 4 |
| total | 9 |

The model was trained for n steps while the performance is monitored (Figure 7). The training was stopped when the model did not improve any further.

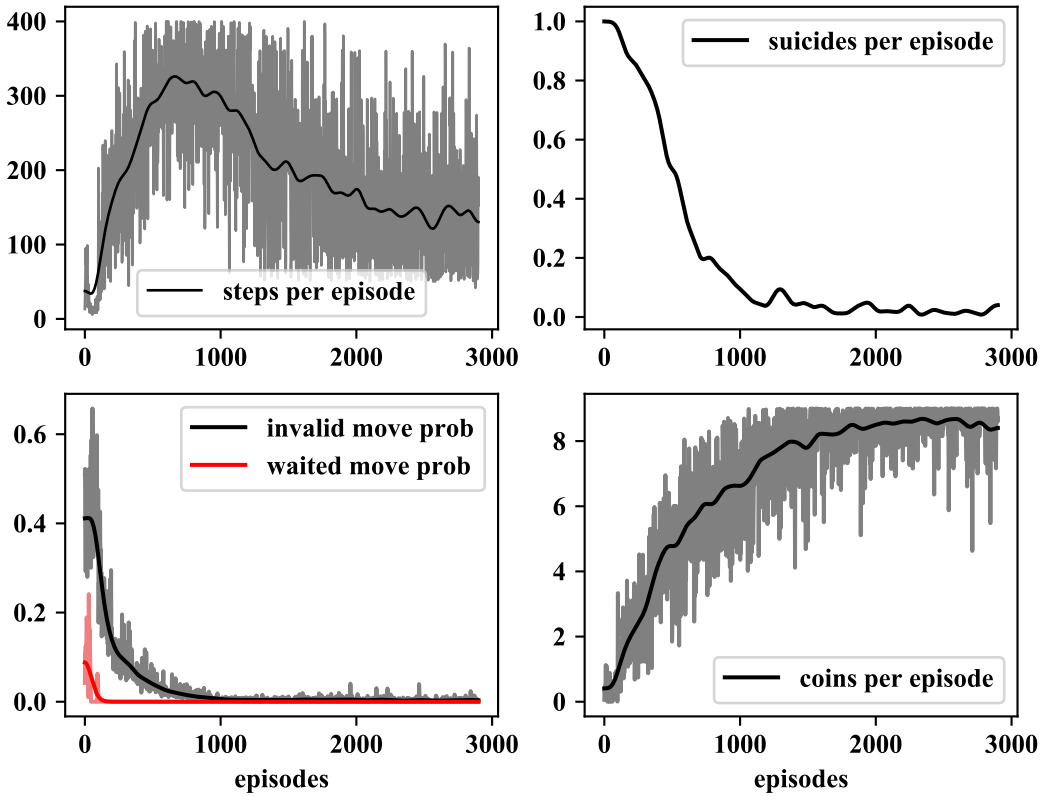



Figure 7: Performance metrics during the training of the decision tree for task 1

The average score trained model is then compared to the random, peaceful and rule-based agent under the same conditions (Table). The Model trained on tasks 2 and 3 was also compared.

Table 7: Comparison of the Performance on task 1

| Agent | average Score (1000 samples) |
|---|---|
| DecisionTree(10k) | 7.86 |
| DecisionTree(1M) | 0.52 |
| peaceful agent | 3.3 |
| random agent | 0.28 |
| rule-based agent | 8.0 |

### 3.4.3 Game with/without opponents

Since the required features and the resulting behaviour is very similar in the last two tasks, only one model is trained for the third task. It should also perform well on the second if and first, if trained correctly. From the available features, 24 features with high performance on the "feature benchmark" were chosen (Table 9). The training was done in three sessions, with different values for reward, epsilon and number of training steps. The rewards for each session were chosen by hand as a reaction to the performance on previous sessions. A decision tree model with a circular

experience buffer with 1M steps and a minimum of 20 leaves per node was used. As a policy, $\varepsilon$-greedy was used with a starting value of $\varepsilon = 1$ which decays with a lifetime of $\tau = 500$ episodes. The final value of $\varepsilon$ in the last training session is $\varepsilon = 0.01$.

Table 8: Assigned auxiliary rewards at the start of the training

| event | general reward $r_t$ | immediate reward $i_t$ |
|---|---|---|
| `INVALID_ACTION` | -0.05 | -2 |
| `WAITED` | -0-05 | -2 |
| `CRATE_DESTROYED` | 1 | 0 |
| `COIN_COLLECTED` | 1 | 0 |
| `KILLED_OPPONENT` | 5 | 0 |
| `BOMB_EXPLODED` | 1 | 0 |
| `KILLED_SELF` | -5 | 0 |
| `GOT_KILLED` | -5 | 0 |
| `SURVIVED_ROUND` | 2 | 0 |
| Agent explores new square | 0.05 | 0 |
| Agent visits old square | 0.05 | 0 |

Table 9: Selected features for task 2 and 3

| name of feature | number of features |
|---|---|
| sum of crates around agent | 2 |
| exploded crates and agents if bomb is placed | 2 |
| distance to closest agent | 1 |
| relative coordinates of closest agent | 2 |
| number of crates around closest agent | 1 |
| distance to next coin | 1 |
| placing a bomb is possible | 1 |
| relative coordinates of closest coin | 2 |
| obstruction level of closest coin | 1 |
| agent position relative to wall | 1 |
| free space in each direction | 4 |
| survivable spaces for each action | 5 |
| total | 24 |

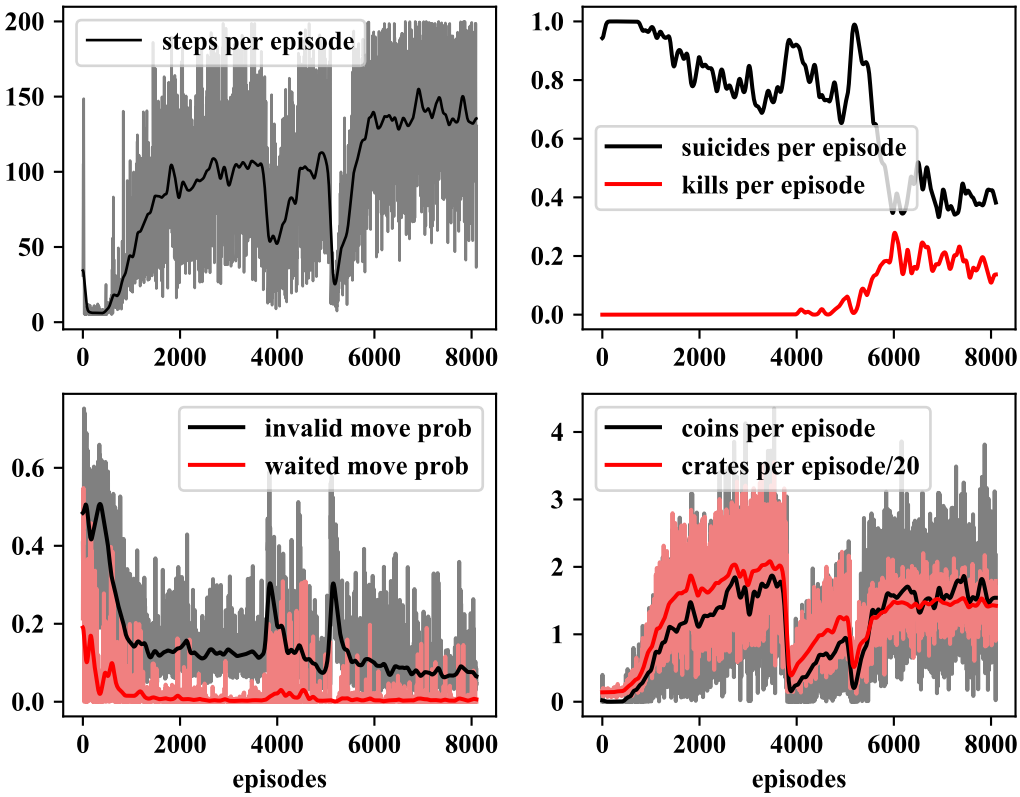



Figure 8: Performance metrics during the training of the decision tree during self-play with 3 agents

Table 10: Comparison of the Performance on task 2

| Agent | average score task 2 | average score task 3 |
|---|---|---|
| DecisionTree (1M) | 2.44 | 2.0 |
| peaceful agent | 0 | 0 |
| random agent | 0.004 | 0 |
| rule-based agent | 8.34 | 3.4 |

# 4 Results of Training Process

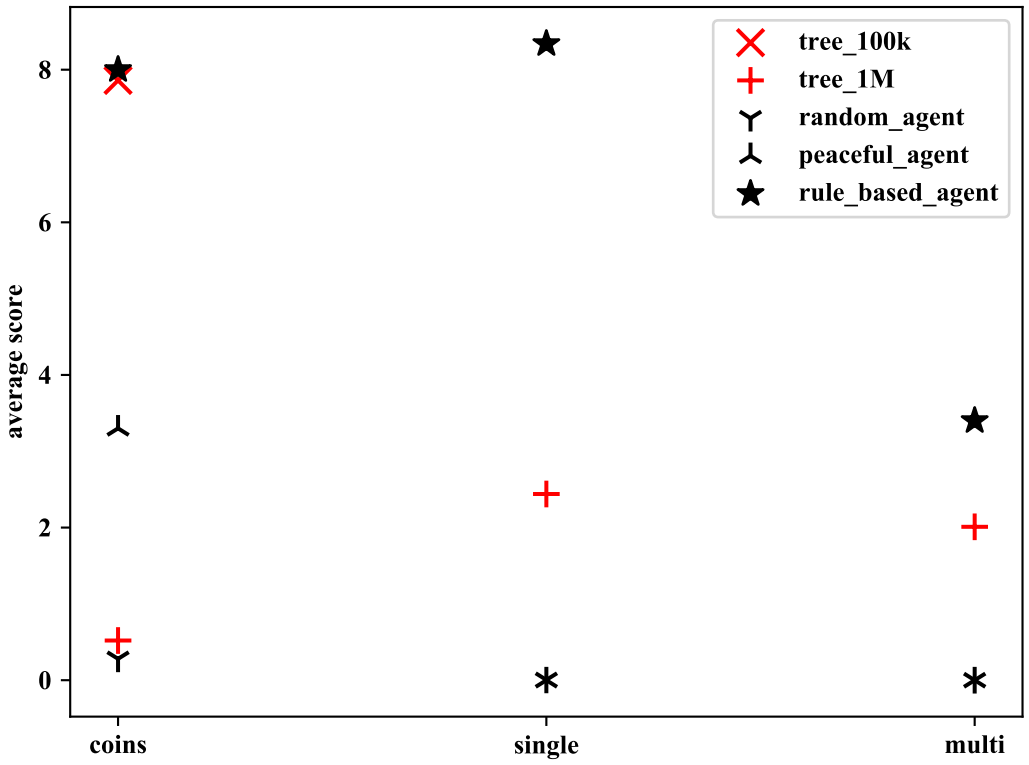



Figure 9: Average score of all agents for the three different scenarios

As can be seen in Figure 9, the performance of the agent trained on task 1 is almost as good as the theoretical maximum of 8 coins per round. Unfortunately, the agent trained on task 3 cannot transfer its abilities to task 2 and 1. Even on the task it is trained on, the reward is significantly lower than those of the rule-based agent. This is likely because of the high suicide rate: The agent gets a lot of negative rewards for doing things wrong, so it tries to terminate the episode and take the -5 suicide penalty as soon as it cannot find any new rewards. Despite this, the agent manages to collect an average of about 2 coins and destroys about 20 crates on average.

## 4.1 Summary

We have created an agent that learns to play Bomberman using reinforcement learning. By choice of the right features and reward fucntion, it is possible for a simple machine learning model, such as a decision tree, to learn the rules of the game and score some points. An important part of the learning is the design of the auxiliary reward function.

# Appendices

## List of Figures



## List of Tables

# References

[1] *(PDF) Applied Logistic Regression.*

[2] L. Breiman, *Random Forests*, tech. rep. (2001), pp. 5–32.

[3] *Epsilon-Greedy Algorithm in Reinforcement Learning - GeeksforGeeks.*

[4] I. Guyon, J. Weston, and S. Barnhill, *Gene Selection for Cancer Classification using Support Vector Machines*, tech. rep. (2002), pp. 389–422.

[5] S. S. Hameed, O. Olayemi Petinrin, A. Osman Hashi, and F. Saeed, "Filter-Wrapper Combination and Embedded Feature Selection for Gene Expression Data", Int. J. Advance Soft Compu. Appl **10** (2018).

[6] *Pearson Correlation - SPSS Tutorials - LibGuides at Kent State University.*

[7] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction* (The MIT Press).

[8] *Tonightâs Starting Lineup For Your Cleveland Cavaliers.... â Cavs: The Blog.*