University of Heidelberg

Final project of the Lecture Fundamentals of Machine Learning

Reinforcement Learning for Bomberman

Florian Ernst and Nick Striebel

Prof. Dr. Ullrich Köthe

March 2021

Abstract

This report explains the operating principles and the development of the agent *Maverick*, that plays the game Bomberman based on reinforcement learning strategies. First, the final concept consisting of a small neural network in combination with hand crafted features is discussed. In addition, discarded approaches and the progress to the final versions are analysed. The final version of our agent manages to score approximately 3.2 points in a game against the three rule based agents that are provided in the frame work of the final project.

Contents

0 Preliminaries

The framework for this project can be found at

`https://github.com/ukoethe/bomberman_rl`.

The code for our agent *maverick* can be found at

`https://github.com/nickstr15/bomberman`

in the sub-folder /bomberman/agent_code/maverick.

To run this agent the packages NumPy and PyTorch are required.

As the project is graded individually we mark who is responsible for the respective sections. The parts written by Florian Ernst are printed in black and the parts of Nick Striebel are colored blue.

1 Introduction

The goal of this project is to implement and train an agent that plays the game *Bomberman* as successful as possible using reinforcement learning.

The rules and exact gaming set-up is documented in the project instruction [KKD21], but for the sake of completeness the essential parts are mentioned here: The agent plays on a 17x17 field against three other agents. Beside the other agents walls and crates are spread on the map. Playing in discrete time steps the agent can decide to move left, right, up, down, to wait or to throw a bomb to attack opponents or to destroy crates using bombs. At the same time he could die by getting hit by an explosion. Coins hidden under crates can be found and give the agent one point if one is collected. 5 points are earned if a bomb placed by the agent kills an opponent.

With this short overview the goal is clear: A good playing agent should learn to collect as many coins and kill as many opponents as possible while avoiding being killed.

To achieve this reinforcement learning is used. This technique and how it is used for this game is the main part of this report.

2 Learning method and design choices

2.1 Drafting of the task

This subsection elaborates the challenges that are coming with the task of creating a reinforcement learning based agent playing the game Bomberman and should motivate our approaches and ideas.

At first it should be mentioned that the game state of Bomberman game with the described setup produces a number of possible game states which is very large. As we are dealing with a 17x17 field we have 289 tiles. The positions of the 113 stones are fixed and 176 free tiles are left. These free tiles can contain different *elements* of the game. It can indeed be free, containing a player, a crate, a bomb with a timer, a player and a bomb, a coin or an explosion. To illustrate how high the number of game states can be we calculate the number for a simple example: Consider a setup with only one player at a random position and 9 coins that are on any other free tiles the number of game-states that can theoretically occur is

$$\frac{176!}{9!166!} \approx 10^{17} \tag{2.1}$$

Here 176 is the total number of tiles, 9 the number of coins and 166 the number of free tiles $(176-9-1)$. Realising that this situation is a lot less complex then the real setup of the game it should be clear that the number of possible game states is too large to predict an action for every possible state, neither there is a chance to see every game state during training.

Another important factor for the success of the project is the time limit. The training of the model needs to be converged before the deadline is reached. This is why some approaches might need to be discarded as can be seen in section 2.2.

Coming from these considerations two crucial goals have to be achieved

1. Find a representation of the game states that is small enough to not break the numbers, while it still needs to allow the agent to distinguish crucial game states
2. For learning a regression model is needed, which is simple enough to be trained in time and which is complex enough to manage the complexity of the game

2.2 Discarded approaches

2.2.1 Linear Model

Our first training model was a linear one. We decided to use it because of the good understandability of it. Furthermore we received the actual learning formulas in the lecture *Fundamentals of Machine Learning*. So this model was a good starting point. In order to use a linear model one has to design features that "vote" for one or more actions. In the end these "votes" are weighted linearly and the action with the highest "vote" is chosen. Furthermore the value of the highest "vote" should approximate the expected remaining points. The latter part is a downside that one could easy miss. In simple words the learning algorithm tries to minimize the difference between the prediction by the "votes" and the actual points. However, this will not work if the features cannot be linearly combined to the expected remaining points. This was a problem that we encountered during our training. For a more detailed analysis see below. Because of this, the linear model was not able do deliver good training results and we had to discard it.

2.2.2 Convolutional neural network

A quickly thrown away, but very interesting approach was the idea of using a convolutional neural network (CNN). The idea was to give the network an image or image like representation of the game state and let the CNN learn the ideal parameters in the training. We tried out 3 different game state representations:

- Providing a RGB image with 17x17 pixels with different colors for each object (for example black for free tiles, red for opponents, yellow for coins,...)
- The second approach was very similar to the first one but just used one channel so the input data could be interpreted as a gray-scale image
- Lastly, we tried to use different channels for different elements. The first channel for example would have been a 17x17 tensor with zeros for free tiles, ones for crates and zeros for stones. At the end we would have used 5 channels
    1. The field (free: 0, crates: 1, walls: -1)
    2. Coins: marked with 1

3. Own agent: marked with 1
4. Opponents: marked with 1
5. Bombs and explosions: marked with a value depending on the bomb timer

Our network had several convolutional layers followed by some fully connected ones. Unfortunately we could not see any improvements in the agents behavior even after long trainings. After doing some research we found the reason: For example [KDW18] uses a similar setting. In the experimental results can be seen, that they needed a minimum of 100.000 training games to see significant learning effects [KDW18, p. 7], even though they had a much less complex gaming framework without coins and only consisting of 7x7 tiles. As we did not use cuda these numbers of training games are not reachable for us and we decided to throw the idea away. Though we still think that this would be the best approach if training time was not limited and that this network would beat any of our agents as soon as it is trained long enough.

2.3 Final model and learning method

As we implemented a CNN for one of our approaches we decided to use it for our final version as well. The main reason for making this choice was that we decided in pre-discussions to use a linear regression model instead of some other possibilities like Regression Forests as this gives us more options in the training progress. Choosing a simple neural network over the linear model seemed to be useful for us as it gives us the opportunity to make use of non-linear functions.

2.3.1 Neural networks

Neural networks are not discussed in the lecture so their operating principle is shortly described:

A simple neural network consists of multiple layers, an input layer followed by hidden layers and an output layer. Each layer has a specified number of neurons. The neurons of one layer are linked to the neurons of the previous one. How these neurons are linked depends on the weights in between. Additionally activation functions to introduce some non-linearity and to change the activation of the neurons can be used. If one considers the special case of only fully connected layers without activation functions, one can see that the neural network simply is a linear model. During training the weights of the networks getting optimized in such a way that the network delivers better results. How exactly this learning works is described in section 3.

2.3.2 Neural network of the agent Maverick

The final neural network that we decided to use is a very simple one: It only consists of the input and output layer with a single hidden fully connected layer in between. Here, fully connected means that each neuron is connected to each neuron of the previous layer via a linear transformation

$$x_i = x_{i-1}A^T + b \tag{2.2}$$

here x_i and x_{i-1} are the neurons of the layer i and the previous one. A is the weight matrix and b is an additional bias vector that is learned.

The input layer has 23 neurons as the state representation is a 1D tensor with 23 entries. The number of neurons in the output layer is fixed with 6 as we have 6 different actions.

We decided to let the hidden layer have 60 neurons. We tried out larger values as well as we tried to use multiple hidden layers, but the use of more neurons/layers only ended up in longer training times. Additionally, the hidden layer uses the rectifier

$$f(x) = \max(0, x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases} \tag{2.3}$$

as activation.

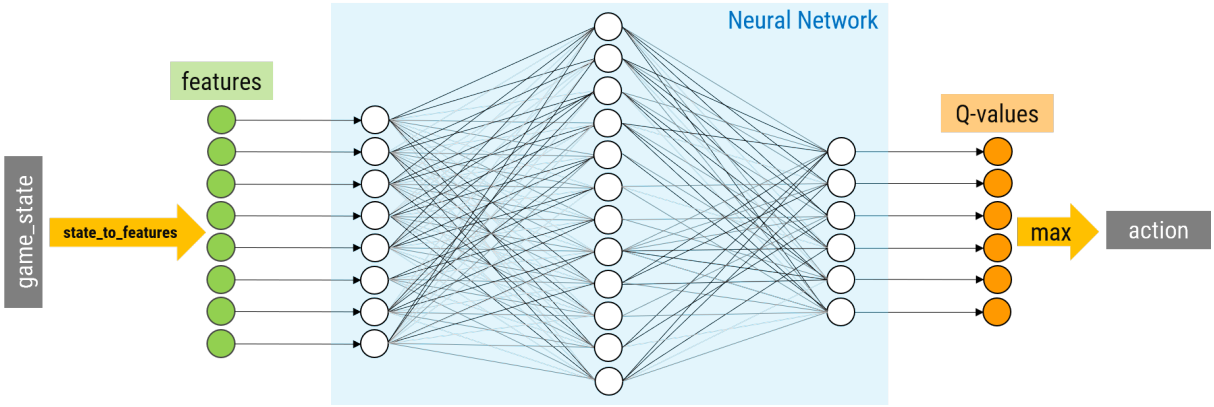

Figure 2.1: *Decision progress of the agent. For better visibility the numbers of features and neurons in the network are reduced.*

2.4 Final state representation

2.4.1 General approach

Since we started our project with a linear model, we had to create features that work in that context. The actual way to the final feature representation is shown in section 3.2. Even though we found out that a linear model might be not as good as a neural network we kept the strategy of using features that *could* be used in a linear model. Of course this is not optimal in many ways. As we discussed above this does worsen the optimal result of our learning process. But there are two major advantages which compensate the drawback.
First, one has to consider that the point of having more states (e.g. if we simply pass the whole field) will always result in a larger computation time. Since the management of the computation time was very crucial we had to use a reduced feature space. So the question was not if we reduce the feature space but how we are going to reduce it.
The second point is the reason why we used exactly the approach of features that could be

understood by a linear model. Of course it might have been possible to use general feature reduction methods like PCA. This method for example was explained in the lecture and it might result in good features, maybe even in better features than ours, but the results would be very hard to interpret by a human. So it would be very hard to interpret the reason why our agent would fail and to correct the bugs causing the behavior, if any. Especially if we consider the fact that the network is pretty much a black box as well. But the primary goal of this project was to *understand* the learning process. Only the secondary goal was to build a very powerful agent. And having comprehensible features enables us to much better understand the problems.

Because of this, we decided to stick with features that could be used by a linear model. A while after we decided to use this strategy we realized that it had another advantage that helped to speed up the process of finding good models. Since we understand the features we were able to create a rule based agent by simply filling a matrix that corresponds to our rewards. This rule based agent was often used to create training data. We estimate that using it reduced the time to the individual convergence by a factor of $\approx 3-5$, by comparing with the purely random generated training data. Furthermore the learn behavior seems to be better. More to this point see Figure 3.4.

The final feature vector consisted of several parts.

2.4.2 Dijkstra search algorithm

Most of the features were computed by a Dijkstra algorithm. The algorithm calculates the *inverse* real distances to all coins and opponents before the search is stopped. Furthermore we look for the inverse distance to interesting bomb positions. In addition to the position and the distance the algorithm remembers the fist step from the starting point. This information is used by the agent to decide for the direction of the next step. Furthermore we estimated that each time we cross a field where a bomb is going to explode in the future we would need two turns because of uncertainty.
The criterion to stop the search early is the following: If we found a coin or a good crate and if the distance is larger than 20 fields (hyperparameter) we break the search. This seemed senseful since the agent needs 20 turns to reach an object that is 20 fields away from the agent (not considering building abbreviations) and the field would change a lot until the agent arrives. So only the general direction to the nearest objects might be useful. The hyperparameter of 20 is not very sensitive since the algorithm will always run until it found anything and the inverse distances at 20 are already very small. Of course there is room for improvement. For example the possibility of forcing a way with bombs is totally ignored by our features. So our agent cannot learn this behavior. Furthermore objects that are not connected to the agent by a direct way, especially opponents, are ignored. Their inverse distance is set to zero if no way can be found.

Interesting bomb positions are defined by two additional hyperparameters: We look for the three nearest crates and the two nearest dead ends. The corresponding feature is the product of the number of crates that would be destroyed and the inverse distance. We restricted the amount of found crates to five in order to achieve a faster convergence. In

practice we think the agent finds very good bomb positions, so this seemed to be fine. However one has to consider that this is still a hyperparameter. An agent that gets the whole field would surly find better bomb positions.

After the computation of this distances we return only the maximal value for each initial step and for each of the three categories (coins, crates, opponents). We keep the product of the crates features as described above. This is a very strong restriction but it delivers good results and it reduces the dimension of the features from 72 to 12. The large version of 72 features did not converge in the period of a whole day of training (cf. Figure 3.3). So we discarded it early.

2.4.3 Features about the local aspects

The other features were calculated by looking at a local region. There are four additional categories:

1. Is it useful to drop a bomb? (1 feature)
2. Will the agent die if he runs in the direction (X,Y)? (4 features)
3. Is a bomb ticking on the agents field? (1 feature)
4. Is an explosion present on the neighboring field (X,Y) the next turn ? (4 features)
5. How high is the remaining possible reward? (1 feature)

The usefulness of dropping a bomb is "-1" if we have no neighboring opponent and no neighboring chest, if we have no bomb, or if the agent would certainly kill himself. Otherwise it equals the amount of destroyed crates. If there is an opponent on a adjacent field the usefulness increases by five. Obviously this feature is very restrictive. If we had more time for the training we would have liked to optimize the bomb placing features more.
Furthermore we have no features that indicate smart moves to kill opponents. This is a large disadvantage from a competitive perspective. But there are that many possibilities to perform tactical moves that it would not be expedient. We think that such moves should be learned by the agent himself and not by a good manual feature design. So if we want a *tactical* agent, we think we would have to use the full feature space. Because of this we did not implement strategy features, only the distance to the other agents. Giving strategy features just feels like constructing a rule based agent, not a learned one.

2.5 Final learning method

We decided very early that we use temporal difference Q-learning. The advantages of Q-learning are that the policy of the agent can be updated without knowing the underlying model of the game. Furthermore it supports off policy learning, meaning that observed game states played with previous/older policies still can be used for training. As our features are not discrete and the feature space for a game with a complexity like Bomberman is very large, the table of the Q-function can not be learned (this is already discussed in

the beginning of this section). To be able to still work with the concept of Q-function, it is learned by regression.

This means that the learning algorithm uses alternating optimization to learn the optimal Q-function (in our case this means optimizing the weights in the NN). The algorithm works as follows:

1. Compute the target response for the reward in game τ at time step t using the current guess for Q
2. Update Q such that the loss of the observed rewards $Q_{\tau,t}$ and the target responses $Y_{\tau,t}$ is minimized

The idea of temporal difference Q-learning is to calculate the expected return using the maximum Q-value for the future step:

$$Y_{\tau,t} = \underbrace{R_{\tau,t+1}}_{\substack{\text{actual reward for action} \\ \text{in time step } \tau,t}} + \gamma \max_a \underbrace{Q\left(s_{\tau,t+1}, a\right)}_{\substack{\text{current guess for Q function} \\ \text{in game state s for action a}}} \tag{2.4}$$

Here γ is another hyperparameter, the discounting factor, which is discussed later.

We started using temporal difference Q-learning as it was easy to implement and to check that it worked. However we adjusted it later and are using n-step temporal difference Q-learning for the final training runs. Instead of using the immediate actual reward the next n actual rewards are taken into account for the calculation of the expected response:

$$Y_{\tau,t} = \sum_{t'=t+1}^{t+n} \gamma^{t'-t-1} R_{\tau,t'} + \gamma^n \max_a Q\left(s_{\tau,t+n}, a\right) \tag{2.5}$$

With n another hyperparameter is introduced. They are discussed in section 3.3.

3 Training setup and progress

3.1 Training of the neural network

Training the neural network works in two steps. The first one is the so called forward pass. Here we let the network do the prediction for training instances and calculate the loss (some difference in the predictions of the network and the *true* values). In the context of reinforcement learning the prediction of the Q-function describes the expected remaining reward. The *true* values are not known, so we have the estimated returns (labelled with Y) that are calculated using the n-step temporal difference Q-learning algorithm described in section 2.5. In the second step the parameters/weights of the networks get updated by using gradient boosting using an optimizer (in this step an update is computed using the derivative of the loss with respect to the parameters).

Implementing this updating algorithm we had two design choices to make: Which loss-function to use and which optimizer to use. We figured out that using the mean-squared-error (MSE)

$$loss = \frac{1}{N}\sum_{i=1}^{N}(Q_i - Y_i)^2 \tag{3.6}$$

works well. Here N is the number of elements and Q_i and Y_i are the predicted and the estimated values for the Q-function. As optimizer we decided to use Adam. The algorithm is in detail described in [KB14], but is shortly described here. The algorithm updates the parameters until they converge. One single update iteration in iteration t works as follows:

1. The gradient of the loss with respect to the parameters θ is calculated: $g_t = \nabla_\theta(loss)$
2. Next the first and second raw momentum m and v are updated using the gradient: $m_t = \beta_1 \cdot m_{t-1} + (1-\beta_1)\cdot g_t$ and $v_t = \beta_2 \cdot 2_{t-1} + (1-\beta_2)\cdot g_t^2$
3. These two moments are biased towards zero (due to there initial values at $t = 0$), so bias-corrected versions are calculated: $\hat{m}_t = m_t/(1-\beta_1^t)$ and $\hat{v}_t = v_t/(1-\beta_2^t)$
4. With these values the parameters are updated: $\theta_t = \theta_{t-1} - \alpha \cdot \hat{m}_t/\left(\sqrt{\hat{v}_t} + \epsilon\right)$

Here m and v can be interpreted as exponentially moving averages. β_1 and β_2 are hyper-parameters which control the exponential decay rates of these values. We had satisfying results using the default values 0.9 and 0.999. Another hyperparameter we used with the default value of 10^{-8} is ϵ, which improves the numerical stability of the update. α is the learning rate (sometimes called step-size) of the optimizer.

Making these design choices there are still hyperparameters that need to be chosen for the update process:
The learning rate (which was just discussed) and the batch-size for the update algorithm.
Looking beyond the update algorithm there are more hyperparameters:
How often to update the parameters, reward-shaping and much more that influences the training. The decisions for these parameters are discussed in section 3.3.

3.2 Feature development

3.2.1 Collecting the coins

We started our project with a "no crates and no opponents" setup. Thus the nine coins were visible and the only task was to collect them. As described above we first tried to just pass the whole field as a feature to a neural network. However we realized soon that it was not a good strategy considering the short amount of training time we had. Because of that we created features that were usable by a linear model. In fact we actually tried to use a linear model for the first week. Our first approach for the features related to the linear model was to simply pass the inverse distance to all remaining coins after the agent did a step or the number -1 if this step is not allowed.
This 36 features do of course not span our feature space, but they should be really good

for this first setup. They consider the fact that we have a lot of symmetries in the system. We can rotate the field by an integer multiple of 90° and still receive a valid state. Furthermore we could mirror the field without leaving the allowed space of possible game configurations. A distance, a scalar, is invariant under this transformations. The fact that we calculate the distances after a first step enables the agent to find the best direction to go this turn. At least these were our thoughts.

We tried three different techniques in order to calculated the inverse distances. The first one was to simply use the 2-norm of the distance vector. However, we found out that this features caused the agent to run against a wall / wait next to it if the next coin was directly behind that wall. This is a logical behavior since the norm or the distance had to become larger in order to reach the coin. This is why we discarded the 2-norm. Another approach we tried was to use the 1-norm. Though, the features calculated this way led to the same problematic behavior. The third idea was to use a breadth-first search. Using the real distance as the distance measure fixed the problem mentioned before. Later we generalized the breadth-first search to work for weighted graphs by using the Dijkstra-algorithm instead.

Of cause we were aware of the downsides of the distance features: The 2 dimensional aspect of the game is not described by the features. It might be senseful not to walk to the nearest coin, but to a place where a large cluster is placed. In the end this first setup can be reduced to a variant of the traveling salesman problem. What is the shortest path that connects all coins. However, we do not need to arrive at our starting point. For this problem no efficient algorithm is known and thus our simple approximation would surely not deliver the perfect result. Though, one has to note that the optimal route is only relevant as long as there are no opponents. Once there are opponents that are collecting our coins as well, it should be advisable to collect the nearest coin since the chances to claim it if we only use simple features are higher.

After some time we realized that the feature space with 36 features for the coin is still to large. Thus, we returned the sum of the inverse distances to all coins after the first step as a first guess for the features. With these 4 features we were able to optimize our model and our hyperparameters. After all 4 features are very few. This optimization resulted in an average number of 5-8 collected coins per run. Obviously it was worse than we expected it to be. Most of the time the agent got stuck somewhere without any obvious reason. Our reduction factor of the Q function at this time was between 0.8 and 0.9 depending on the individual run. Our first idea was that the endless loop behavior would be an effect of a to short training period. However, now we are confident that this was not the case. We tried training sessions with up to 1,000 games without success (later on we were able to achieve very good results with 200 games or less). In fact the playstyle of the agent became worse. After the 1,000 games it went into the endless shivering loop almost immediately.

Our other ideas of punishing loops via negative rewards and a memory of the recently visited places did not result in the wanted behavior as well. Instead the agent created larger loops and adapted a playstyle that would be expected in snake rather than in bomberman.

He always tried to avoid his own path even though we only punished him if he was more than 7 of the last 20 rounds on the same field. Actually neither the punishment nor the memory were necessary to solve this simple setup.

The first working feature setup was rather a random encounter than a planned one. We found it during our tries to make the features more significant. One try in order to achieve this was to encode the four features in a hot one manner. Thus we computed the four features that were either the sum of the inverse distances or -1 and calculated the argmax. In a next step we assigned to all positive entries in the feature array the value 0. After this we assigned to the index that was the argmax the value +1. This feature setup worked much better than the ones before, so we kept testing with it. For one configuration it surprisingly worked. We did not assign a one but the remaining number of coins. After this suddenly all endless loops vanished in almost all training configurations we tried before. We needed a few days for the interpretation of this effect. Now we think that the reason was a way to large reduction factor. Because of the lecture *Fundamentals of Machine Learning* we thought that the reduction factor was only a mechanic to keep the Q-function from diverging, however it also influences the expected remaining reward. With a reduction factor of one we try to approximate the whole remaining reward by the Q-function. But if there is no opponent and if we play good enough this will *always* be the remaining number of coins. But if one combines the features in a linear manner there is no dependence of the resulting Q-function on the remaining coins. Because of that the best way for the agent to approximate the linear model (or the small neural network) to the expected remaining points was indeed to do nothing. In this case the expected points are constant the negative reward of a step. Since the agent always decides for the optimum of the Q-function there had to be a kind of shiver. Always waiting was not a solution because single bad actions are intrinsically suppressed.

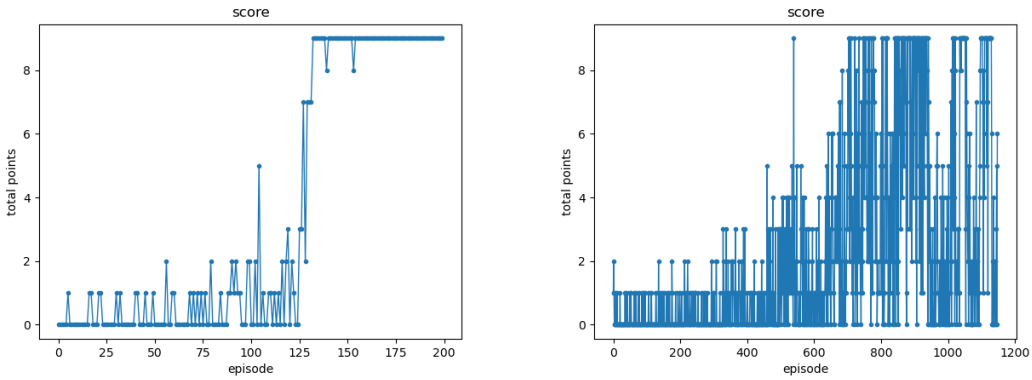

Figure 3.2: *Comparison of the training with high and low reduction factor. Left: Very good results with* $\gamma = 0.6$. *Right: The optimum is not stable* ($\gamma = 0.85$).

By reducing the reduction factor from 0.85 to 0.6 we were able to mostly eliminate the shivering and the endless loops. This makes sense:
Our features allowed the agent to estimate only the points of the next coin. So we approximately predict only the next 8 turns (instead of the 70 turns to complete the

game). After 8 turns our reward is reduced by an effective factor of 0.017, so it can effectively be neglected. To conclude our finding: If we use bad features we have to use a smaller reduction factor than if we used good features and in our case we should use $\gamma = 0.6$ in order to get good results (see Figure 3.2).

3.2.2 Destroying crates

Once our agent was able to reliably collect the revealed coins we changed the training setup. From this point on the coins were hidden behind crates. So the agent had to bomb the crates in order to find and collect the coins. However, in the second setup were still no opponents present. The biggest issue in this second step was to keep the feature space small since the number of objects goes up by a factor of more than 10. Furthermore it was impossible to simply return the inverse distance to the next crate if the agent goes in direction (X,Y) since the bomb positions are not equal. The agent looses time if he blows up only one crate per bomb. But he looses time as well if he searches too long for the best bomb position. The best features if one decides to keep the distance-feature-concept would be most likely the distances to all places next to a crate and the number of crates that would be blown up at this position. If all those features would be passed to the network, the agent could learn the best trade off between distance and number of destroyed crates. However, the features space in this case is very large. We tested this feature combination (in truth we passed only the five best bomb positions (later)) paired with data generation via a rule based agent. Even after more than 10,000 games (a whole day of training) the results were very bad.

Figure 3.3: *Training of the agent with inverse distances to all coins and to the best five bomb positions. The generation of the training data was stopped after 9,000 games. Usually the agent plays much better at this point in the other training runs. Here the agent merely collected 2-4 coins and killed 1-2 peaceful agents that randomly ran into his bombs.*

Due to this we were not able to use this large feature space. Nevertheless, we found a really good approximation. The product of the inverse distance and of the number of destroyed crates was a good equivalent to the inverse distance in the previous step.
Another problem was the fact that the agent had to place bombs in this step. Therefore "bomb features" became necessary and the pathfinding algorithm had to take care of paths that crossed future explosion fields. In addition to the need of reducing the feature space and considering the bombs in the search algorithm it was necessary to optimize the feature calculation. The amount of needed training games went up from 200 to 10,000 in order to achieve good results (mediocre results after 2,000).

Because of this we changed the feature calculation as follows:
First of all we replaced the breadth-first search by the Dijkstra-algorithm. This enabled us to use a weighted graph. Furthermore we decided to store the first step in the heap queue. As a result each object was only found in the step that brings the agent closest to it. However, only one search and not four searches (one per step) were necessary.
The next modification was to increase the distance counter of the Dijkstra-Algorithm by 2 (instead of 1) if the path crossed a field that was threatened by a bomb. The reasoning for this is the uncertainty that comes with crossing those fields. For example it might happen that one has to wait.

After this we decided to implement some break criteria. The first one is based on a boolean variable that is set to true once a crate or a coin in has been found. If this variable is true and if we have exceeded a certain distance (we used 20 steps), the Dijkstra-algorithm is left early. This break criterion enabled us to run the feature calculation code approximately 2 times faster and since the feature calculation is the bottleneck in our training, the total training time went down by approximately 0.5 as well. The quality of the features and results did not seem to suffer under the break criterion. After all the game is most likely to change a lot in the next 20 turns. At least if there are opponents.
The second method to speed up the feature calculation in this second setup was to pre-filter the bomb locations. We decided to only consider bomb-positions next to a crate. Furthermore we realized that the best bomb positions - neglecting the distance - are dead ends. After all there are usually the most crates that can be destroyed by bombs. Having this in mind we defined our pre-filter criterion as follows:
"Find the two nearest dead ends and the 3 nearest positions next to any crate, that are not already chosen".

We decided to base the bomb evasion only on local information because every dangerous field (= a field threaded by a bomb) that is more than 5 fields away from the agent will be safe before the agent reaches it. So we did not use the previous Dijkstra-search for the following features. The first additional feature was effectively the number of crates that would be destroyed if a bomb would be dropped at the position of the agent. In case that no crate was on an adjacent field we returned the value -1. Later we returned -1 also in the case that the agent would surely kill himself by his own bomb.
The next feature was a (short) breadth-first search that calculated whether the agent could escape the dangerous fields if he walks now in direction (X,Y). Depending on the answer the feature was -1 or 0.

The next four features described whether a bomb would explode on a adjacent field. These features were meant to prevent the agent from running into his death since the previous four features were only $\neq 0$ if the agents field was in danger in order to reduce the computation time.
The last bomb evasion feature was the negative amount of turns until the bomb on the current field would explode, zero if the field was safe.

Since we found out that the missing information of the remaining points was a problem, we decided to return the maximum value of remaining points as a feature, too.

3.2.3 Playing against opponents

In this third training setup we trained the agent with opponents. Therefore we modified some of the features. However most of the features from the previous setup stayed the same. First of all we added four features that described the inverse distance to the opponents. Again, we used only the maximum of the inverse distance in each direction in order to keep the feature space small. Furthermore we increased the feature that indicated the effect of a bomb on the agents field by 5 if an opponent was on a neighboring field.

Obviously these features are not sufficient for a strategic opponent hunt, but they are good enough to stay alive. We considered the possibility of creating explicit strategy features but discarded this idea soon because we realized that this would not be the essence of a learning algorithm but the design of a pseudo rule based agent. Instead we analyzed the behavior of the agent after a training with our features. Most of the time we used a rule based agent, created by a manually designed linear model, to create the training data. This rule based agent had an attractive value for each opponent in order to produce (interesting) conflicts. However, the trained agent always learned to run from the opponents, even in the training with the peaceful agents. Though, the effect was stronger when learning against the given rule based agent. Once most of the crates were destroyed and almost all coins were collected, the agent ran into a corner and waited. If an opponent walked to him he ran away. But this happened only after the coins were distributed, before he *was* fighting the opponents. Considering his bad features, he did a good job, though. So ultimately the agent *learned* that his features are not good enough for winning by killing opponents but they were good enough to win by collecting the coins faster. We think that this behavior is actually very smart. So we think to make him fight actively, better features are needed. If we had more time one try would have been to pass the old features and a local 5 x 5 area as plain field. Since the most relevant fighting information should be local this features could have solved this behavior. But the remaining time was too short to start this training.

3.3 Hyperparameters

During the training we tried several combinations of hyperparameters. They can be categorized into three general fractions:
Model-related, feature-related and training-related.

3.3.1 Model-related hyperparameters

As we discussed above we chose an neural network as the underlying model. After some testing we found out that one hidden layer with *60* neurons was a good trade off between needed training time and quality of the results. As described above we used a squared loss and the Adam optimizer. During the training progress we tested several values of the learning rate $0.1 > \alpha > 0.0001$. However, we found the native value of the optimizer (0.001) to work best. For the other learning rates we received worse results after the training.

3.3.2 Feature-related hyperparameters

In our features we used some hyperparameters as well. The main reason was to speed up the computation time of the feature vector. Therefore we restricted the number of found dead ends to two and the number of found additional crates to three. Furthermore we decided to break the Dijkstra search early once a good destination was found and if the search radius had exceeded 20 steps.

3.3.3 Training-related hyperparameters

The last category of hyperparameters are the parameters of the actual training algorithm. Since we decided to use an experience replay based training strategy, we had to choose a buffer-size and a batch-size. For the size of the experience buffer we tried several values between $1,000$ and $4,000$. This equals a memory of 2.5 to 16 games for the small value or for the larger value 10 to 67 games, respectively. Once we even tried a buffer size $10,000$ steps but we realized that the results did not become better. Only the learning speed went down. In the end we chose a buffer size of $2,000$ steps for the final training.

In the first half of the project we used a simple variant of experience replay with only one batch size. The batch was randomly chosen from the experience buffer. For this setup we tried batch sizes between 50 and 100 elements. We realized that a batch size of 100 elements slowed down the training progress and did not improve the convergence speed and the quality of the result much. Here we decided to use a batch size of 50 most of the time.
In the second half we modified the learning to prioritized experience replay. Since then we used a batch size of 120 elements for the randomly chosen batch and created a sub batch containing the 50 elements with the highest residual found in the first batch.

Furthermore we used a variant of the linearly diminishing epsilon greedy training strategy. We started with an epsilon of 0.5 and ended with a value of 0.05 after 90 % of the games in the training. For this factor we tried values between 70 % and 100 %. After this amount of steps we kept the ϵ-value constant. Furthermore we modified the behavior of the decision made if the ϵ-criterion holds. We sometimes used a rule based action instead of a random action. The chance for this was an additional hyperparameter. We tried values between 90 % and 0 %. For the final training we used a value of 90 % if the ϵ-threshold was above 10 % and 0 % otherwise. More to rule based actions in section 3.5 on training strategies.

3.4 Reward-shaping

An important approach for speeding up the training process is the usage of auxiliary rewards. We decided to use larger rewards than the one in the game, so we could work with integer values. This is definitely not necessary but makes it easier to estimate the effects of the single rewards compared to the others.

We developed our final version in three steps:
Only coins, coins hidden by crates and complete game. We started the task of finding good auxiliary rewards with a game field that only contains our player and the nine randomly spread coins. As the goal was to collect the coins as fast as possible we introduced a negative reward (-1) for every action that agent takes. An even bigger penalty is given for an invalid action (-10) to teach the agent not to take an impossible actions. Forcing the agent to collect the coins and making him recognize that this is a very good behavior we started with a reward of +500 per collected coin. This approach gave us good results as soon as our features and model was selected accordingly well. One thing that was already mentioned: Our agent sometimes got stuck in loops. To prevent this we tried to penalize this behavior. However, as explained before, this just leads to larger loops in the training.

Continuing with the second task the rewards had to be extended and adjusted. Now the agent had to learn that he has to destroy crates to reveal the coins. Due to this new task the agent needs to learn dropping bombs in good positions while not killing himself.
Making him recognize good bomb positions the agent earns a reward of +33 for each crate he destroys. Here we used a trick that worked very well: Instead of handing out this reward after the crates were destroyed we hand out the reward directly after the agent drops the bomb. So he earns a reward of +99 if he drops a bomb at a position in which the bomb will destroy three crates once the bomb explodes. Of course in the complete game at the end it can happen, that the crates are destroyed by an opponent before the own bomb explodes, though this was not a problem for the game without opponents and we recognized that this situation that two agents attack the same crates occurs not that often.
To prevent the agent from killing himself we penalize his death with -700 reward points. This set of rewards delivered good results for the single agent mode and after again adjusting the model and features the agent managed to collect most of the coins.

Tackling the task of playing the game in the complete setup we did not changes a lot. We only added a reward of +500 for killing an opponent and adjusted the reward for collecting a coin to +100 as we wanted to have the same ratio as in the real game mode. We thought about the idea to apply a similar trick for placing a bomb next to an agent as we did it for the crates, but decided not to use it due to two reasons: The number of crates is fixed so the total reward for destroying the crates has an upper limit, while in theory the opponents can survive till the end of the game. Here we fear that the rewards could explode. The second thought going into this decision: The agent should learn to stay alive and should play safe-oriented (if he dies too early he cannot collect any points). If we had more time to train and test this definitely would have been one of the next try-outs.

Studying the behavior of our agent it seemed like the agent preferred to destroy crates instead of collecting the coins. We tried to counteract this behavior by lowering the reward per destroyed crate to +10, but overall this ended up in a game performance which was much worse.

The final rewards for training that we figured out to perform well are displayed in Table 3.1. The table clarifies the general approach we took: To clearly distinguish between good and bad actions/events.

event	**reward**
any action	-1
invalid action	-10
collect coin	100
kill opponent	500
drop bomb that destroys n crates	$33 \cdot n$
get killed/self-kill	-700

Table 3.1: *Final auxiliary rewards to speed up training.*

As mentioned in the part of the feature development our agent is not learning any tactical moves for fights with opponents. We did not start any tests or training into this direction, but for this it definitely would be good to have fitting auxiliary rewards as well. Maybe then the previously discussed idea of giving rewards for bombs placed close to other agents would be useful.

3.5 Training strategy

Having a good training strategy is essential for good results. After the training is finished the agent should have seen enough game states to handle every situation that occurs. Of course, as we worked ourselves through the three tasks our training strategy varied and we optimized it.

Starting with the first task of only collecting the coins we decided to use an ϵ-greedy policy. So the agent chooses an action randomly with a probability of ϵ and explores the game- and action-space. With the probability of $1 - \epsilon$ the agent decides according to the current model, optimizing the policy towards the local optimum (exploitation). We decided to use a linear decreasing value for ϵ followed by a phase of a small constant ϵ value at the end of the training (see section 3.3.3). The idea behind this constant part is, that at the end of the training (hopefully) the best local optimum is found and this then should be optimized as good as possible. For the coins-collection-task this worked very well and we had good results after 200 training games.
One issue that we had was that the agent sometimes got stuck in a loop and started vibrating between two tiles. We managed to reduce this problem by fixing a bug in the training algorithm: We updated the parameters for the used decision model in every time step. Due to this, the agent escaped the loops in single games as the parameter updates forced him out of them. This is why the training results looked much better than the

scores during testing. After we fixed that, the parameters got optimized in every time-step but the model/network only gets updated after every game. Unfortunately, we did not manage to prevent the problem of loops completely.
Another big step towards better results was achieved by two adjustments at once: We reduced the discounting factor and additionally adjusted a crucial part of the training algorithm: Normally if the upcoming state is a terminal state the expected reward for this state is 0 (in the calculation of the expected return). We decided to change this and set the terminal states to the previous ones (*old game state = new game state*). This is not the best solution but regarding that playing a full game only every 400th state is terminal that seemed to be okay. Later we recognized that the problem probably was, that at the end of a game due to the structure of the implementation not all rewards are handed out correctly. So instead of our impure approach tackling the problem from this side would have been better. We kept our version as we did not want to change such a crucial part shortly before the deadline.

With this training setup we were prepared very well and in principle we did not have to change that much for the final version. For task two we basically kept the same strategy and only had to play a larger number of games to get satisfying results. Here we also tried to build up our agent,first training him on the coins-task and then on the setup with crates. This did not work well as the agent learned to ignore every feature beside the coins in the first phase and we discarded this approach for the rest of our trainings.

For the third part we also tried to keep the same strategy as before, but did not receive pleasing results. Here the properties of our features came in very handy. As we could interpret the features in such a way that a linear model can work with them as well we created a rule based agent that decides coming from our features by multiplying the feature vector with an hand-crafted matrix. We tested this agent and it played very well against the rule based agent provided in the framework of the project. We decided to change the behavior in the exploration part to sometime choose a rule based determined decision instead of a pure random decision to show the agent useful moves. This method of course has the risk to introduce a bias but we decided to accept this as the results are much better, as it can be seen in Figure 3.4.

An important decision was also the selection of opponents in the training sessions. We mainly had three options: Playing against the peaceful agent, the provided rule based agent or against our own agent. After testing a little bit we quickly found out that the rule based agent was just too strong and our agent died too fast in the games. At the end the best results were delivered by playing against the peaceful agent. Although our agent only played against this non-bomb-dropping opponents the behavior in testing games against more aggressive opponents is still quite good.

Our final training strategy summarized: Use ϵ-greedy in combination with rule-based determined decisions, playing against peaceful agents. Of course there is still potential to optimize this, but at some point we had to fix the strategy to complete the project before the deadline.

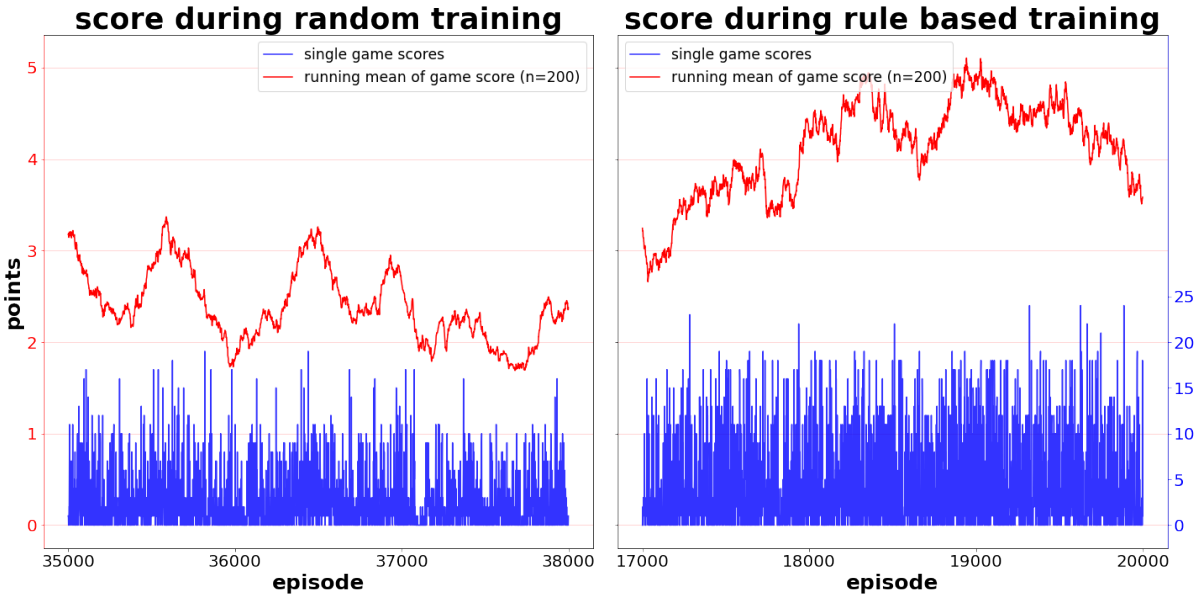

Figure 3.4: *Comparison of the last 3000 training episodes against three peaceful agents (please consider the two different y-scales). Left: Training with random decisions during exploration. Right: The decision during exploration is performed by rule based agent (in 9 out of 10 cases), the average score is significantly higher.*

4 Experimental results

In the following section we are going to summarize the observations we made during the project. Most of them have already been mentioned above so we are going to keep it short.

One of the first things that we found out is that a linear model is much weaker than a neural network, but it converges much faster. So it is a question of time whether or not one has to use a linear model - or whether or not one has to interpret the results. Of course we already knew that a linear model is inferior, but this project was a nice illustration on this topic.

Furthermore we had the problem of an agent caught in an endless loop. We worked on different layers of the training process in order to reduce this problem. We found two aspects to be most important.
On the one side we changed the new game state to the old game state if the new state would have been none. This happened if the game ended. One might think that it would be not problematic to just skip the last step as we did before, but in this case the agent will never learn that collecting the ninth coin is *good*. So he does not know what to do after he collects the eighth coin and starts to shiver.
One the other side we used strongly simplified features that were not able to describe the total remaining points well. As a reason of that we suspect that it was almost impossible to combine those features to the expected remaining reward. The use of a strongly reduced reduction factor solved this problem. So we realized that the reduction factor is related to the quality of the features.

Interestingly both changes were needed. The correction of only one of it was not enough to stop the shivering.
A third observation correlated to the shivering was that it is not good to update the used Q-function during a running game. If the function is updated after every step, the agent will break out of the loop eventually and thus not learn that this behavior is counterproductive. So the learning progress is slowed down. Instead it is better to update a copy of the working model and change the copy to the used model after the game.

Another result we found out is the fact that one has to choose a training strategy that matches the goal of the project and the deadline. We were not able to use the (probably) best training strategies because of the early deadline. We were not aware before that neural networks need that much time to converge for such simple tasks. It is interesting to see that the use of simplified features and data generation by a rule based agent was able to speed the training process that drastically. Even if the end result suffers it is more important to have a converged network.

Additionally we have realized that the agent seems to *realize* that his features are not good enough to fight the opponents in a direct confrontation as described in section 3.2.3. So he learned to flee from other players. This effect was observable with every used opponent, except the random one. He destroyed himself before the agent could see him.

For a final test we played our agents once against three random agents and once against three rule based agents (3,000 games each). The results are plotted in Figure 4.5. Regarding, that our agent does not learn to fight, a score of approximately 3.2 points in a game against 3 fighting agents is not bad (only 9 points are available in the form of coins).

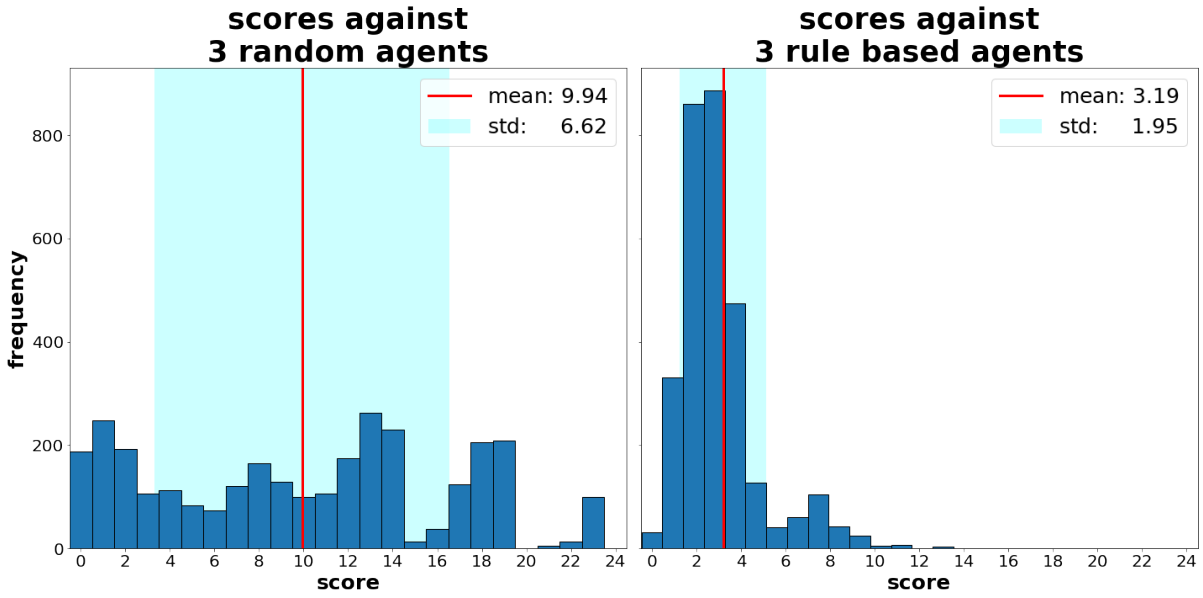

Figure 4.5: *Final test of our agent. Left: The scores in 3,000 games against peaceful agents. Right: The same test against three rule bases agents.*

5 Outlook and Discussion

Our agent is not playing perfectly by far. The weaknesses are mentioned in the respective sections. here we just want to sum them up and give an outlook on what we would have liked to improve.

At first it should be mentioned that we used a rather conservative approach to tackle this project. We used hand crafted features that can be interpreted in a linear context. Also the NN only consists of one hidden layer to introduce some non-linearity. The reasons for this were already explained: We wanted interpretable features so that we could identify mistakes more easily. Of course this might limit the quality of the final results, but doing it this way we think, that we learned a lot more, as we could understand what is happening.

Nevertheless, we still think there is room for optimization in our approach. If we would have more time we would have made a larger test series on different hyperparameters-settings. This might give us better results in shorter computation time. The biggest deficit of our agent is, that he has no fighting strategy. This could not be learned as we did not provide any features for these situations. Beside this we faced some situations in which the agent simple does not move, even at the beginning of the game. These two problems would have been the next issues we would have liked to solve.

In general we are content with our results, especially with the learning progress we had towards the end of the project. Mainly the time factor withheld us from trying out more concepts. For example the idea of providing the whole field of the current game state to a CNN is still very present in our minds. This also would have been an approach we would have tried out to learn fighting strategies by providing something like a 5x5 field surrounding the agent.

6 Feedback

As we already mentioned in the previous section we learned a lot during this project even if the result is not perfect, but this learning proceeded foremost towards the end of the project. We needed more than two weeks to had setup of features, rewards and model that worked at all. Here we would have appreciated some small tips to start the project that provide some information on promising concepts regarding the model and features. Of course this is a critical balancing act as the project might become to easy. As mentioned quite often it would have been nice to have more time for this project, especially regarding that at the end of the semester there are a lot of other exams and deadlines. Maybe it is possible to shift the lectures on reinforcement learning to December, so more time would be available to try out more concepts. Still we learned a lot during this semester and the project, on no account should be replaced by an exam or something similar. One learns to work in a team and to become acquainted with new issues.
The setup and environment provided for Bomberman is quite good already. We think the important issues/optimization suggestions are already discussed in discord.

References

[KB14] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. URL: `https://arxiv.org/pdf/1412.6980`.

[KDW18] J. G. Kormelink, Madalina M. Drugan, and M. Wiering. *Exploration Methods for Connectionist Q-learning in Bomberman*. 2018.

[KKD21] Koethe, Kruse, and Draxler. *Final Project: Reinforcement Learning for Bomberman*. 2021.