Ruprecht-Karls-Universität Heidelberg

Fundamentals of Machine Learning

Final Project

Reinforcement Learning in Bomberman

Authors
Christopher Brückner
Computer Science B.Sc.
Ubeydullah Ilhan
Computer Science B.Sc.

Supervisor
Dr. Ullrich Köthe

March 25, 2019

Preliminaries

The public Github repository containing our code can be found at:
https://github.com/chbridges/bomberman_agent

The code is based on the libraries NumPy, scikit-learn and LightGBM.

1 Learning method and design choices

1.1 Representation of the game states

Written by Christopher Brückner

The arena of the given Bomberman framework consists of 289 tiles, 113 of which are walls, i.e. only have a single state across the whole episode. The remaining 176 tiles are free tiles which can adopt the following states:

- Free
- Containing a player
- Containing a coin
- Containing a crate
- Containing a bomb, represented a timer (default: 4) decrementing with each step
- Containing a bomb and a player (possible after placing a bomb and waiting)
- Containing an explosion, also represented by a timer (default: 2)

This gives us a total of 14 different possible states per free tiles and a total of $14^{176} \approx 10^{201}$ states when considering all of them at once. We can neglect the wall tiles in our approximation, as they are constant. Of course, due to the rules of the game, merely a fraction of these states can be reached: There is a maximum of 4 players, each player can only place a single bomb, the positions of explosions are bound to the position of the bomb that created them, there is a maximum of 132 crates (default value) and 9 coins.

However, as our agent, i.e. our regression model doesn't know the rules of the game, this gives us an upper border of states we have to deal with within the scope of our project.

In task 1, the arena consists only of 1 player and 9 coins, reducing the amount of states immensely – there are $\binom{176}{9} \approx 10^{14}$ possible positions for the 9 coins when initializing the round, which stay constant until collected by the player, and 176 possible positions for the player.

However, this amount of states is still way too large for creating a table representing values for all possible (s, a) tuples, whereas s is the corresponding state and a one of the 6 possible actions left, right, up, down, bomb and wait. It is extremely unlikely that an agent will visit the same state twice, even across several hundred episodes.

In front of our lie fully observable Markov states. However, according to our approximation, using all data the game provides is very inefficient: Calculating rewards for every possible (state, action) pair would require days, if not even weeks or months of training. Therefore we need to extract good features in order to reduce the amount of states as much as possible without having a too large trade-off in terms of performance.

First of all, instead of representing a state with all absolute coordinates of the player and the coins (and even more targets in the subsequent tasks), we use their coordinates relative to the player, i.e. we subtract the player's position from the target's position. This allows us to neglect the position of the player itself, which amounts to 176 possible values.

Next, we only take into consideration only the nearest target of a type relative to the player, i.e. the coin with the least distance to the player within the scope of task 1. To achieve this, we use a modified version of the look_for_targets() function implemented in the simple agent which returns us the absolute coordinate of the next target instead of absolute coordinate of the first step towards the target. As the relative position of the nearest coin is represented by a tuple (x,y) with x, y $\in$ [0,15], this reduces the amount of states to only 225.

To prevent the agent from walking against the wall, we also include the adjacent tiles to his position in our features, with 0 representing a free tile and -1 representing a wall, as given in the arena array.

In total, our states in task are represented by the following feature vector relative to our agent:

(top tile, bottom tile, left tile, right tile, nearest coin x, nearest coin y)

In task 2, we pursue a similar approach. We append to the feature vector the relative coordinates of the nearest crate, the nearest bomb and the nearest dead end, analogous to the nearest coin in task 1. We chose to include dead ends as they will either lead to more points when placing a bomb there (larger density of crates), or lead to the agent's death after placing a bomb.

f Furthermore, the first four features (adjacent tiles) now represent not only walls, but every tile the agent is not supposed to enter or try to enter: The features are 0 when they are free, and 1, when they are occupied by a wall, a crate, a bomb or an explosion.

As a final feature we add a flag showing whether or not the agent has placed a bomb. 1 means that he has placed a bomb, encouraging movement and discouraging trying to place another one.

In total, our feature vector in task 2 looks like this (square brackets representing coordinate tuples):

(top, bottom, left, right, [next coin], [next crate], [next dead end], bomb flag)

We also tried negating the coordinates of the nearest dead end when the bomb flag is set to 1, tricking the agent into moving into the opposite direction after placing a bomb, however this approach didn't seem to work.

In task 3, we simply added the relative coordinates of the nearest player to the tuple, i.e. our final agent observes 15 features of the current state of the game before making a decision.

1.2 Learning algorithm

Written by Christopher Brückner

As we observed in 1.1, it is impossible to record and predict the rewards for every single (state, action) pair possible in the game. Therefore we need to approximate the Q-function returning the expected reward by means of regression.

We chose the Q-function, or action-value function, over the regular value function, as it will more accurately predict the optimal policy based on what the agent has learned so far; due to the imminent danger of nearby bombs, Bomberman requires predicting good strategies for many different situations, which is why sufficient knowledge about the outcome of actions and subsequent actions is vital.

The Q-learning algorithm is given by the following update rule

$$Q(s_t, a_t) \leftarrow (1-\alpha)\, Q(s_t, a_t) + \alpha \left(r_t + \gamma \, \underset{a}{max} \, Q(s_{t+1}, a) \right) \tag{1}$$

where t is the current time step in the episode, s_t and a_t are the state respectively the executed action during this time step, s_{t+1} the subsequent state after executing action a_t, r_t the reward returned for this action.

The update rule can be transformed into the following, computational more efficient form

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \delta_t \tag{2}$$

where

$$\delta_t = r_t + \gamma \, \underset{a}{max} \, Q(s_{t+1}, a) - Q(s_t, a_t) \tag{3}$$

is the temporal difference error.

The hyperparameters $\alpha, \gamma \in [0, 1]$ can be described as following:

α is the learning rate, i.e. it influences the rate of to what extent the current learned Q-function will be adjusted to the most recently recorded rewards. $\alpha = 0$ means no learning, $\alpha = 1$ means completely overwriting the previously learned approximation. In theory, the policy π should converge against the optimal policy π^* for decreasing α or for constant, very small α, if π^* is existent.

γ is the discount factor which influences the weight of the subsequent optimal policy added to our current reward. As each subsequent reward is dependent on the final reward, the expected total reward R at the end of the episode decreases exponentially for $\gamma < 1$ and is given by

$$R' = \gamma^t R \tag{4}$$

Therefore, a lower discount rate will possibly lead to a solution needing fewer time steps, but will also possibly fail, as the agent can't predict subsequent rewards, especially when the episode consists of many time steps.

Another possible algorithm would have been state-action-reward-state-action (SARSA). It is given by the update rule

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_t + \gamma\, Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right) \tag{5}$$

Whereas Q-learning is an off-policy learning algorithm based on tuples (s_t, a_t, r_t, s_{t+1}), updating the Q-values based on the estimate maximum subsequent reward, SARSA is an on-policy learning algorithm based on tuples $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$, updating the Q-values based on the policy the agent currently follows. SARSA possibly leads to more accurate results after sufficient training, but as we already have to deal with a massive amount of data, it is not efficient in our case.

1.3 Exploration/Exploitation

Written by Christopher Brückner

As for the the exploration-exploitation trade-off, we decided to use a ε-greedy strategy. We introduce a third hyperparemter $\varepsilon \in [0, 1]$ in order to determine to which extent our agent explores or exploits the game. The agent will execute a random action with a probabilty of ε and stick to his learned policy with a probability of $(1 - \varepsilon)$. $\varepsilon = 0$ resembles a greedy strategy, i.e. the agent only exploits and strictly follows his policy, and $\varepsilon = 1$ resembles a random-walk strategy, i.e. the agent only explores and executes random actions.

Within the scope of task 1, choosing a high ε already achieved good results. As our agent only has to collect coins and works with their relative coordinates, a high degree of exploration allows him to quickly explore which actions are beneficial in order to receive a large reward, i.e. approach the closest coin.

However, this strategy is not sufficient in tasks 2 and 3. As the agent needs to place bombs in order to score, he will quickly blow himself up when randomly placing a bomb at an inconvinient position and when randomly waiting or walking against a wall after placing a bomb. Therefore we relied on a diminishing ε-greedy strategy, i.e. we trained the agent multiple times with decreasing ε. In other words: During the first iterations, our agent explores the next immediate actions and in the best case learns how to destroy crates and escape bombs. During the later iterations, our agent puts his focus on exploitation, i.e. he sticks to what he previously learned and explores actions around his current policy in order to improve it.

According to Kormelink et al., the Max-Boltzmann strategy delivers the most accurate results and the highest win rate in Bomberman after sufficient training iterations; however, as Max-Boltzmann is more difficult to implement and takes more training in order to outperform other exploration-exploitation strategies, we decided to stick to our previous approach.

1.4 Regression model

Written by Christopher Brücknerl

In the beginning, we tried to approximate the Q-function using simple linear regression, adjusting the individual weights instead of the Q-values. The weights were given by the following update rule:

$$w_i \leftarrow w_i + \alpha \left(r + \gamma \max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right) \cdot f_i(s_t, a_t) \qquad (6)$$

whereas f_i are the features representing the state and w_i are the corresponding weights. However, this approach failed, as the weights diverged and returned NaN (not a number) after only around 20 training iterations.

We henceforth used ensemble methods, as suggested in the lecture: regression forests and gradient boosting. Unfortunately, during our first tries, our implementation of the Q-learning algorithm was wrong; neither the random forest regressor nor the gradient boosting regressor returned satisfying results. However, we quickly decided to utilize gradient boosted trees, as they delivered the same results as the random forest, i.e. constantly walking against the wall, while computing significantly faster.

We furthermore discovered the LightGBM library, whose implementation of the gradient boosting regressor performs better than sklearn's implementation in terms of speed and accuracy.

In order to approximate a Q-table, we use sklearn's MultiOutputRegressor returning a #actions-dimensional, i.e. 6-dimensional vector with reward predictions for each (state, action) tuple.

Within the scope of task 1, we only let the agent execute the actions up, down, left and right. There is no need to place bombs or wait, as there are no imminent dangers and the agent is only supposed to navigate through the arena as quickly as possible, so we simply did not teach him the actions yet in order to get faster results. We added the remaining two actions in task 2.

2 Training process

2.1 Rewards

Written by Christopher Brückner

We chose to give our agent the following rewards at the end of each step for the occuring events:

COIN_COLLECTED	100
KILLED_OPPONENT	100
SURVIVED_ROUND	100
CRATE_DESTROYED	25
BOMB_EXPLODED	10
BOMB_DROPPED	-5
INVALID_ACTION	-10
WAITED	-50
KILLED_SELF	-50
Everything else	-1

We clearly distinguish between good and bad actions. We give a lot of points for collecting coins in order to make the agent quickly learn that he has to approach coins for a high reward. As our agent is not good at surviving, we give him the same amount of points for killing an opponent as for collecting a coin, so that in doubt he will search for safer ways to score. We deduct points for executing invalid actions, i.e. walking against walls or trying to place a second bomb, in order to encourage movement towards free tiles. We furthermore deduct points for waiting in order to discourage doing nothing, when not in immediate danger like being caught between walls and explosions. The reason for deducting 1 point for every other action, i.e. valid movements across free tiles, is that we want to encourage quick navigation – e.g. our agents earns a slightly bigger total reward when he collects all coins in fewer steps.

Within the scopes of task 2 and 3 we also deduct points for dying – this way we want to encourage escaping the bombs' explosion radius. We give the same reward for killing a player and collecting a coin, so that the agent learns playing safe instead of constantly running into danger when there are other means of scoring points.

2.2 Experience replay

Written by Christopher Brückner

One possible approach for implementing Q-learning is storing all (s_t, a_t, r_t, s_{t+1}) tuples collected during the training. Here, s_{t+1} is the subsequent state after executing action a_t in state s_t, and contains an index, i.e. a pointer to the corresponding row in the Q-table. However, due to the large amount of states in our case, indexing them is highly inefficient.

We solved this issue by saving each variable of the tuples in its own matrix. The number of executed training rounds, i.e. the size of our dataset, is given by M.

observations is a $M \times N$ matrix with N being the number of features, depending on the task.

rewards is a $M \times 6$ sparse matrix, containing the reward returned for the execution of one of the 6 possible actions, the remaining 5 columns being 0.

last_ actions is an M-dimensional vector indexing the nonzero columns of rewards, which is significantly faster than reading the correct column using NumPy's nonzero() function.
Our approach directly using SciPy's sparse matrices unfortunately failed due to a lack of time and due to having 2 non-sparse matrices anyway.

Q is a $M \times 6$ matrix predicting the rewards for all 6 actions in the corresponding state.

Each row tuple (s_m, a_m, r_m, Q_m) corresponds to a tuple (s_t, a_t, r_t), and the subsequent tuple, i.e. $(s_{t+1}, a_{t+1}, r_{t+1})$ corresponds to a row tuple $(s_{m+1}, a_{m+1}, r_{t+1}, Q_{t+1})$, $m \in M$. This allows us the calculate the Q-values for each step iteratively at the end of each episode.
Here, we make use of NumPy's arithmetical operations, which give us a significant speed boost over the standard Python arithmetical operations.

We don't draw a line between mutliple saved episodes – the Q-value of the final step of an episode is dependent on the first step of the subsequent episode. This gives us an error rate of $\frac{1}{400} = 0.0025$, which should not have a huge effect on our calculations.

Unfortunately, for big M - "big" meaning only few hundred rows – the training process becomes quite slow. Therefore we tried to reduce the dataset in size.

First, we tried saving only the most recent 400 entries, i.e. the last episode. Sometimes, our agent actually performed quite well, scoring our highscore of collecting all 9 coins in only 39 steps. However, there is a huge randomness factor, as each played episode is heavily dependent on the preceding one, even though he should play according to a better policy each time.
Our next approach was saving the most recent 10000 entries, i.e. the last 25 episodes. This gave us a bad trade-off between training speed and performance.
Moreover, we were unable to recreate our regression models, as they were all fitted to larger datasets before saving the data and reloading the game.

Therefore we simply sticked to saving all data, resulting in long training times, but also having a reliable model. Due to the long duration of the training sessions, it is vital to choose good training parameters in order to make it efficient: 100 training rounds are futile if the agent constantly walks in circles or against walls, collecting hundreds or even thousands of redundant steps. Another argument encouraging an ε-greedy strategy, as it allows us to easily fine-tune the agent's behavior by exploring further states and actions close to his current best policy, leading to more diverse data.

We also implemented a cleaning function that removes duplicates from the Q-matrix before fitting the regression model to it, giving us a slight speed boost.

3 Experimental results

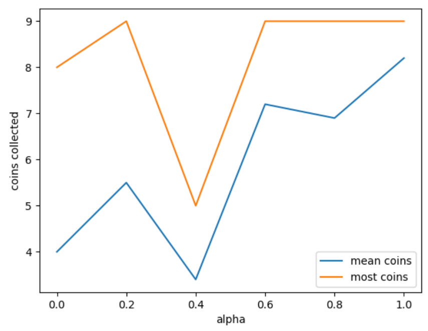
3.1 Tuning the hyperparameters

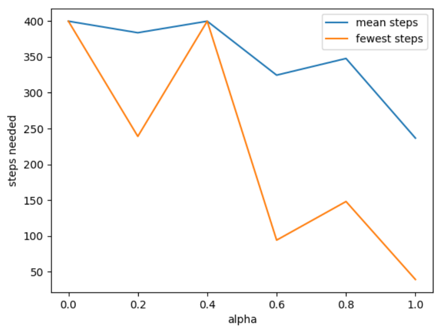
Written by Christopher Brückner

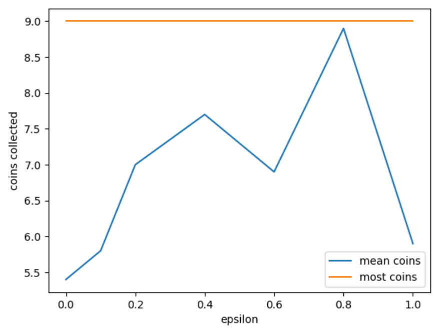
In the following, we explore the performance of our agent in task 1 using different parameters. In each diagram, our agent was trained in 100 iterations using constant parameters $\gamma = 0.95$ and, depending on the experiment, $\varepsilon = 0.5$ respectively $\alpha = 0.75$. He was thereupon tested in 10 games using $\varepsilon = 0.1$. We chose $\gamma = 0.95$ as we wanted to keep the final reward prediction as high as possible, while encouring quick navigation.

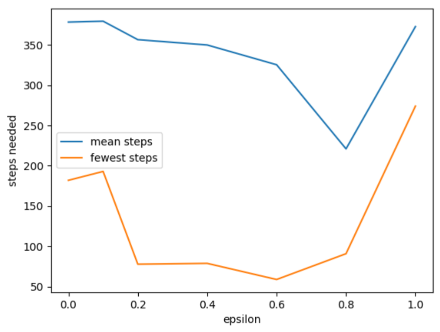
Due to the high randomness of the game, 10 rounds are not overly representative; however, they give a general idea about how well the agent performs with only little training, taking less than 10 minutes on the used system.

The following four diagrams depict the performance of our agent when adjusting α and ε in regard to the collected coins per episode and the steps needed to do so:









The first two diagrams shows that there might be a bad trade-off between the next immediate reward and the subsequent optimal reward; the agent manages to collect more coins when taking the most optimal learned policy into account, i.e. our expectation that the Q-function outperforms the value function is confirmed. With large alpha, our agent performs way better in task 1 with only few training rounds.

The very good performance for $\alpha = 1$ is most likely a coincidence due to a fortunate dataset – we could not recreate this performance after another training session with the same parameter.

The latter two diagrams state that the agent performs best after training with a balanced exploration/exploration-ratio and slightly better with large ε. As stated before, a high degree of exploration is beneficial within the scope of task 1 – still, exploitation is vital for good results.

Incidentally, we already chose relatively good parameters during the early development of our code – as the initial values $\alpha = \varepsilon = 0.75$ yielded satisfying results, we thereafter used them throughout the project. In task 2 and 3, we let ε converge towards 0.

3.2 Collecting coins

Written by Christopher Brückner

As observed in 3.1, using a simple ε-greedy strategy with a good choice of hyperparameters can already yield satisfying results when the task is collecting all coins in an empty arena. In order to improve these results, we use a diminishing ε-greedy strategy, as discussed in 1.3. We therefore trained our agent according to the following strategy: We set $\gamma = 0.95$ and $\alpha = \varepsilon = 0.75$. After 100 rounds, we lower ε to 0.5 and after the subsequent 100 rounds, we further lower ε to 0.25. After 300 total training rounds, we test the agent with $\varepsilon = 0.25$. The first 10 test rounds yielded the following results:

Round	Coins collected	Steps needed
1	9	141
2	9	120
3	9	157
4	9	98
5	8	400
6	8	400
7	9	185
8	8	400
9	9	119
10	9	269

Collecting a mean of 8.7 coins with a mean of 228.9 steps was satisfying enough for us. When tested with $\varepsilon = 0.05$, the mean collected coins reduced to 7.07, but he still reached good scores such as collecting all 9 in only 53 steps.

The main problem is that our agent sometimes gets stuck in a loop, i.e. he walks back and forth on the same two tiles, especially when the next coin is far away - until a random actions brings him on the right track. Therefore we still need exploration to a small degree after training in order to break. This issue becomes less grave in the subsequent tasks, as the agent has more targets (static crates and moving enemies) and has to change his movement pattern after placing a bomb, which he learned not to do in this task.

4 Outlook

Written by Christopher Brückner

Our main problem with the project was getting started. Even though there are lots of example codes to be found, it was rather difficult for us to figure out, how we can apply reinforcement learning and regression at the same time. This made it difficult to debug the code in the initial phase, which is why the choice of features is definetly one of the weak spots in our code - this is the main issue we would put our focus on if we had had more time.

Furthermore, we heavily relied on the knowledge we earned in the lectures and exercises. This knowledge is definetely sufficient. However, as we explained in 1.4, there are libraries reaching better, i.e. faster and more accurate results than e.g. scikit-learn. We were not able spend much time on discovering the full potential that Python offers for reinforcement learning.

Moreover, our training process surely isn't optimal. More training rounds creating a more diverse dataset might be helpful, for example by training the agent against other students' agents, leading to sequences of states we don't discover when training against the simple agent or oneself. Fine-tuning the hyperparemeters for different tasks is another issue we could not resolve, as training took much time.

The latter issue might be resolved by better code optimizations, which is difficult to do, as the project requires simultaneous work on a multitude of aspects.

Interestingly, the students we talked to about the project could all solve task 1, but struggled with the subsequent tasks. Even though the project was fun to work on, it seems like it was rather hard due to its size when compared to reinforcement learning examples on the Internet, i.e. simpler versions of Bomberman (smaller arenas) or simpler games in general. More hints about the selecton of features might be useful in the future.

Exchanging agents for training without sharing their codes is impossible, which is why we suggest more hardcoded agents such as the simple agent next time in order to improve results and prevent plagiarism.