Fundamentals of Machine Learning

Report: ”Reinforcement Learning for Bomberman”

Elias Arnold, Luca Blessing

April 8, 2019

Contents

0 Remarks

General

As agreed with Mr. Köthe we submitted the agent code one week after the official deadline due to an injury followed by a surgery on Elias' shoulder, which made him quite handicapped for the last couple weeks.

Furthermore, we would like our solutions and report not to be seen as separate parts, but one whole, on which we both, Elias and Luca, worked in close cooperation as a team. Elias did more work on our code and Luca wrote more of the report. But since we always checked each others work and discussed all ideas together, we don't supply authorship notes in our report and want to be evaluated/graded as a team. We also just need to pass this project and the grade won't have any influence for us. Please keep that in mind, even if we put a lot effort into coming up with a good solution for the agent.

Another note, is that this is the first time for both of us, to work with neural networks. Therefore a big part of our project work was testing different parameters, getting to know how to use deep learning and gaining experience in what will work for this game setup and what doesn't. We did train our model on a CPU, because in the end we hadn't enough time to parallelize our training and didn't have a GPU at hand.

Run the Agent Code

The only requirement to make our agent code run is an installed version of `tensorflow` which can be installed using `pip`. If one likes to test the agent in the repository that is using a regression tree and a PCA dimension reduction one needs also to have `sklearn` installed.

Final Agent for the Competition

To be honest the final agent, which we submitted on April 1st, works not as good as we have hoped. The code for this model can be found in `agent1_neural_state1`. It is able to stay alive sometimes, destroy crates and collect some coins every now and then or even kill another agent more or less by accident, but the performance could be better. For this agent we used Deep Neural Q-Learning, even if we also tried other approaches to which we will point out in the report.

Because we wrote this report after the submission of our final agent, we found some errors in the model code used for training and also made some small adjustments, e.g. in our state representation. We used this corrected version for our tests in section 3. It also turns out that the additional training, as well as the correction of the errors, provided us with a much better agent, who now plays quite well against the simple agents and wins many games. Of course we can't expect you to accept our new agent to play in the tournament, since it's two weeks after the official deadline. Nevertheless, we would be pleased, if you will at least have a look on it and test its performance. The code to this new model/agent can be found in our GitHub repository at `agent_code/agent2_neural_state2` and we will also submit it along with this report.

Repository

The URL to the repository is `https://github.com/n1ce3/bomberman.git`. The repository contains our agent code as well as our other attempts (different state representations, using regression trees for Q-learning...) to create a good agent. They all can be found in the directory `agent_code`.

1 Method and Model

1.1 Reinforcement Learning

Tabular Q-Learning The reinforcement learning method we use is Q-Learning, where the agents policy is determined by a value Q which depends on a specific state-action pair. In our case the agent always has 6 actions to choose from, `'UP'`, `'DOWN'`, `'LEFT'`, `'RIGHT'`, `'BOMB'` and `'WAIT'`. Its state at time step t is denoted by $s_t \in S$. For tabular Q-learning the state space S is finite and all state-action pairs have a specific Q-value $Q(s_t, a_t)$ representing the expected (future) reward. The agents policy is then found by choosing the action a_t having the highest Q-value for the given state s_t.

After every step of the game the agent receives a reward r_t and finds himself in a new state s_{t+1}. With the obtained information (s_t, a_t, r_t, s_{t+1}) the Q-function will be updated.

$$Q(s_t, a_t) = (1-\alpha)Q(s_t, a_t) + \alpha\left(r_t + \gamma \max_a Q(s_{t+1}, a)\right) \tag{11}$$

The learning rate $\alpha \in [0,1]$ indicates, how fast the agent neglects old Q-values (higher α means more weight on current r_t and Q). The discount factor $\gamma \in [0,1]$ assigns a certain importance to future rewards and influences the decision making of the agent. The idea here is, that after enough steps the Q-values converge and represent the true, expected rewards for a specific action in the given situation and therefore the agent finds the best policy by performing the action with the highest Q-value: $a_t = \mathrm{argmax}_a Q(s_t, a)$.

Exploration-Exploitation Methods The problem with only using the randomly initialised Q-table, which is referred to as pure exploitation, is that these values don't represent the true rewards for each action. Hence, the agent follows an initial policy and maybe gets to a local maximum, but from there doesn't find another, probably much better policy. So we introduce a random element, where in some percentage of the steps, the agent chooses a random action to explore new policies and finds new states. This percentage of random actions is denoted by ϵ.

What we used, is a diminishing ϵ-greedy exploration method, where in the beginning $\epsilon = 1$ and then decreases over the course of multiple episodes until it reaches a minimum. This leads to high exploration in the beginning, to experience a lot of states, but higher exploitation in the end of training, where the Q-values are expected to be more and more accurate. We tried different methods of decay, like linear or exponential decay, but finally stayed with the exponential diminishing ϵ, since it yielded the best results.

Another method we tried, was a Max-Boltzmann exploration. Here each action is assigned a probability, measured by how good or bad its outcome will be. When exploring, the model chooses according to these probabilities, which should lead to exploration without too often killing oneself (which is a problem when using the ϵ-greedy method, as will get clear in section 3). The implementation can be found in `agent_code/agent5_simpleCoinState`. But this method somehow didn't work well and we already found better results with the other methods, which is why we neglected it later on.

1.2 Trying Different Models

First implementation Our first idea was to find a good state representation leading to a fairly sized state space S and then calculate the exact Q-table for all state-action pairs and it's corresponding Q. The states have just 8 dimensions: the 4 tiles next to the player, the direction of the nearest coin, if dropping a bomb is possible, a danger level defined by the bomb timer and the direction of the nearest bomb.

We use the JSON format to save the Q-table actually as a dictionary within a dictionary, assigning every state six actions, each with its own Q-value. The states are described by 8 letter long strings, one letter for each dimension. The exact definition of how a state is described can be found in `agent4_QTablecallbacks.py`. This way we would have up to $5^4 \cdot 4 \cdot 2 \cdot 4 \cdot 4 = 80000$ states, but since a lot of states will never occur, e.g. all neighbouring tiles are walls, the final Q-table will be much smaller.

This, for example, leads to an agent quite good at collecting all coins in the first task, but it clearly doesn't lead to the best policy. Our agent only chases after the nearest coin (as we tell him to) other than coming up with a more clever strategy. To improve this, we could tell him where all the other coins are, but this would result in a much bigger state space and a huge Q-table, which is not easy to handle. And it doesn't work well, if we add crates and opponents to the arena. Hence, we neglected this simple idea of just using a table with direct mapping of state-action pair to Q-value.

Using regression models Since taking more information into consideration makes the state-space too big and our agent has to carry around a huge Q-table. This requires a lot of memory and the agent needs to experience every possible state and explore all actions multiple times until every Q-value is somewhat accurate. We can avoid this problem by using a regression model, which gets the state as input array and outputs a 6-dimensional array containing each action's Q-value. This should save us some memory and we don't need to explore each and every state to predict quite well Q-values.

Random Forest Regressor `sklearn.ensemble.RandomForestRegressor` is the first regression model we implemented. We tried this model only for the first task, where there are only coins and no crates or opponents. The states are defined using the arena layout. We iterate over each tile and append its type to a list: 0 if empty, 1 for a coin and -1 for the players position. We do not consider the walls, since they always remain the same.

As will be described in more detail in subsection 2.2, we didn't come to good results and hence tried to speed up the learning by doing a dimension reduction/transformation using `sklearn.decomposition.PCA` (Principal Component Analysis). But before seeing any improvement here, we ran out of time and neglected this approach in order to put our focus on the usage of Deep Learning.

1.3 Deep Q-learning Network

Part of our following state representation as well as other ideas in our model are based on a paper about Q-learning in Bomberman. [1]

State Representation Our third approach is a neural network implemented using the `tensorflow` library. Here we chose a different state representation, since we use this model for all three tasks. This refers to the code in `agent1_neural_state1`.

We define and use the `getNearest(pos, obj)` function, which needs the players position `pos` and the objects positions `obj`, from which to find the nearest one. It returns the distance to the nearest object, normalized by the diagonal of the arena, and also a list with 4 values, each 0 or 1. This list is a hint for the agent, where to go in order to reach the nearest of the specified objects. In contrast to our earlier approaches, we didn't take into account if the agent can or cannot drop a bomb, since he should learn this on his own, when being punished for choosing an invalid action.

To keep the state-space small, the latter $4 \cdot 49 = 196$ entries of our state consist of a sub-arena, 7×7 tiles surrounding our player, where every tile has 4 associated values.

- Type: wall, empty or crate $(-1, 0, 1)$
- Danger Level $\in [0, 1]$
- Coin: yes or no $(1, 0)$
- Player: empty or opponent$(0, 1)$

It's sized 7×7 in order for the agent to be able to see his explosions. The danger is either calculated by $1 - 0.17 \cdot (t + 1)$, with the bombs timer t, or 1 for an explosion. The first 15 entries of the players state are given by the return values of the `getNearest`-function, first the nearest of all coins, second the opponents and last the crates. These should serve as hints for the agent to know what is outside his field of view. The exact computation of our player's state is done in the `getCurrentState`-function in `callbacks.py`.

After finding some errors in this agents code, we created a corrected version in `agent2_neural_state2`, which also has a little different state representation. The 15 first entries are the same, but we slightly changed two of the four values assigned to each tile of the 7×7 sub-arena. The danger level is negative for enemy bombs and positive for our agents bomb and we also added the agents own position.

- Type: wall, empty or crate $(-1, 0, 1)$
- Danger Level $\in [-1, 1]$
- Coin: yes or no $(1, 0)$
- Player: opponent, empty or self $(-1, 0, 1)$

[1] http://www.ai.rug.nl/~mwiering/GROUP/ARTICLES/ICAART_EXPLORATION_BOMBERMAN_2018.pdf

Model The model we finally used is a convolutional neural network (CNN) with a single hidden-layer of 550 neurons. The network is fully connected, meaning each feature of our 211-dim. input array, representing the current state, connects to every neuron as well as every neuron connects to each entry of our 6-dim. output array, containing each actions Q-value. Hence, the term "Deep Q-learning Network" (DQN). After a lot of research on reinforcement learning, or Q-learning respectively, and some testing with more layers and more/less neurons, we found this configuration to be useful. The activation, optimizer and loss function can be looked up in our code at `model.py`. Since we're not too experienced in Deep Learning yet, the reasons for using exactly these are that we found them in other examples while researching on the internet and also didn't have too much time to play around and test many different algorithms for their performance.

2 Training Process

2.1 First Implementation

States and Q-table This subsection relates to the directory `agent4_QTable`, which can be found in our GitHub repository. We use the explicit Q-table approach, where we thought of the JSON format to be the most suitable storage. The reason is, we don't have to initialise a `numpy-array` or `pandas-dataframe` with all possible states. As explained in the first paragraph of subsection 1.2 some combinations, e.g. all neighbouring tiles of our player are walls, aren't even possible. This would make no difference in learning, because these states are never reached and therefore their Q-values are irrelevant. But we would either have to think of and sort out each impossible state beforehand or we have to carry around a big table from the beginning. Hence, we use a JSON-file, to which in `act(agent)` we initialize/add a state only if an agent finds himself in it for the first time and in `reward_update(agent)` calculate/update the Q-values according to the received rewards.

Disadvantages The problem with an explicit Q-table is the necessity of the agent to find each and every state multiple times, in order for the Q-values to converge and therefore yielding the best policy. This is why we only consider the four tiles neighbouring the player and additionally supply hints, e.g. for coins and bombs. As mentioned before, after enough training this method finally provides an agent good at collecting coins. But even this task he doesn't solve in the most clever way, since he lacks the information about all other coins.

Eventually, despite all problems and weaknesses of this first approach, we gained good insight and practice about the foundation and limitations of applying Q-learning in its simplest form.

2.2 Random Forest Regressor

States This paragraph relates to the directory `agent3_tree`. The states here were computed in form of arrays with entries for every tile of the arena, which isn't a wall, in order to train `sklearn.ensemble.RandomForestRegressor`. Each tile/feature can be either 0 if empty, 1 if it contains a coin or -1 if our player is on this tile. This way we will have feature arrays with 176 features to train our model with and let it predict 6 Q-values, one for each action.

Model The model is defined in `model.py`. Each time `reward_update(agent)` is called, we add a sample (s_t, a_t, r_t, s_{t+1}) to the models memory. Our rewards are defined when the agent gets initialized in `callbacks.py` and are chosen in a way that the agent learns to collect coins, destroy crates, kill opponents and in general encourage active behaviour. After a specified number of episodes, the model gets trained using a function called `replay(self)` in `model.py`, where we randomly collect a batch from the models memory. Given an input array for state s_t, the regressor predicts an array with the Q-value of each action. In `replay(self)` for each sample the Q-value for the action a_t chosen in state s_t

is updated according to equation (11). Using the updated output arrays containing the desired Q-values as targets, we fit the regressor.

Disadvantages We trained our random forest model trying different parameters like the number of trees, leafs, etc. We also played around with the learning rate α, the decay of ϵ over the course of our training, but we couldn't come to reasonably result in an acceptable time. Because of this we added a PCA-transformation for our state arrays. This way the feature space would be smaller and hopefully our training more efficient. But eventually, after another bunch of trainings using different parameter sets, we still had an awful agent. Thus, we decided to focus on the other model, a neural network, which we implemented at the same time and already showed some progress.

2.3 Deep Q-learning Network

The code, to which this paragraph relates, can be found in `agent1_neural_state1`. It is the one used to train the agent we handed in for the tournament.

Advantages One characteristic, that makes using a Neural Network above a classic regressor a bit more sensible, gets clear in the training process. Even if we just add a little more data to the trainig set, the regressor needs to be trained with all of the data again, leading to a lot of fitting time needed later on. The DQN, on the other hand, can be either be initialized with a pre-trained model, or once train a newly set up model, and from there on only be provided with a new batch of training data, from which it improves on top of the old model. This means we safe the agony of having to train the whole model with all of the data over and over again.

States Our first attempt for a state representation was like the one in subsection 2.2, except that each tile had 4 associated values, like the ones described in subsection 1.3 for the 7×7 sized sub-arena. This representation leads to a state array sized $4 \cdot 176 = 704$. At first glance, this doesn't seem too much, but after training a huge amount of episodes over the course of days, the model didn't reach any point of acceptable results for the third task, where crates and opponents are present.

The next idea was to just make our players field of view (or "sub-arena", as it is called in our code) smaller to reduce the amount of states, in hope of a faster learning process. But this reduces also the information on coins, crates and opponents, since the agent won't know anything about the events outside this sub-arena. Probably because of this lack of information, our agent did indeed perform quite well in task 1 and 2, but he clearly wasn't good enough to face task 3, where opponents are present.

The final composition of states we used for our Network is already explained in subsection 1.3. The idea is to keep the state space small by only considering a 7×7 sub-arena around the players position, but also provide him with some information about nearest coins, bombs and crates.

Rewards Since this is the final model we used, we also want to shed some light on the rewards we use as incentives for our agent to collect coins, kill opponents and in general follow an active strategy. The reward used for each of our models can be found inside `callbacks.py` in the respective directory. But to get an insight into those used in this final model, they are listed in Table 1.

MOVED_LEFT	-1	BOMB_EXPLODED	0
MOVED_RIGHT	-1	CRATE_DESTROYED	40
MOVED_UP	-1	COIN_FOUND	0
MOVED_DOWN	-1	COIN_COLLECTED	100
WAITED	-1	KILLED_OPPONENT	350
INTERRUPTED	-1	KILLED_SELF	-300
INVALID_ACTION	-10	GOT_KILLED	-100
BOMB_DROPPED	-2	OPPONENT_ELIMINATED	700
SURVIVED_ROUND	200		

Table 1: Rewards used to train the Deep Q-learning Network which we submitted

The -1 on all movements or if the agent waits should cause an active behaviour and therefore lead to events where the agent can gain some of the big rewards. Also by having a high reward on destroying a crate, the agent is motivated to use bombs, leading to revealed coins and free tiles. Of course the highest rewards are given for collecting coins and killing opponents, while killing oneself or getting killed is highly undesirable. After we submitted our code and started to make some new trainings in order to have some experimental data, we realised some mistakes. We mixed up some of the events and, for example, gave a reward for OPPONENT_ELIMINATED even though this means that an opponent killed another opponent, without our player being involved.

Training In order to train our model, we used a diminishing ϵ-Greedy exploration-exploitation strategy, as explained in subsection 1.1. We tried different types of decay for our ϵ, like linear or exponential. After multiple tests, we came to the finding, that the exponential decay is most suitable. In this case most of the exploration is done in the beginning, but our agent still experiments enough later in training to hopefully not get stuck in a one-sided strategy.

Like with the Random Forest Regressor, we train after a specified number of episodes, 100 episodes in our final training. This is done inside the `replay`-function in `model.py`. Here we compute the target Q-values using the (s_t, r_t, a_t, s_{t+1}) samples, a learning rate $\alpha = 0.6$ and a discount factor $\gamma = 0.95$. Then we run the optimizer with a batch of 10 samples, randomly chosen from memory, which is afterwards deleted. This will be repeated until the whole memory buffer is empty. After another 100 episodes we will have gained enough new data and the model gets optimized again.

The batch size is set to 10 and the data also gets randomly selected from a memory buffer, because the training data should contain data with low correlations in order to achieve a good fit/optimization. On the other hand the batch can't be much smaller, since the model would need a lot more time for fitting and iterating over the whole memory.

Parameters The learning rate α, discount factor γ, the rewards given for different events, the batch size and even the state representation can probably all be chosen differently and improved to complement each other and yield better results in training. As mentioned in some cases, we tried a lot of different approaches, always keeping in mind the impacts this can have, e.g. when adjusting the learning rate α: making it bigger to put focus on newly gained experiences, where the agent eventually made a better move and gained more rewards, or making it smaller to distribute over more samples and keep singular events from disturbing the agents learning.

3 Experimental Observations

Since we submitted the code to our agent before having to hand in our report, we did some testing only afterwards in order to gain some more insight, resolve some problems and find possibilities for improvement. This doesn't imply that we did not also try our best at different approaches, ideas and parameters before, but the plots you will see in this section are from the testing done after the submission of our agent. Some errors we only found in this period of further tests, e.g. in choosing the rewards or in our code. A quite big mistake we found, is in `getCurrentState`, where the 5 values of our agents state for the nearest opponent are falsely assigned with the nearest coin. This reduces the information given to the agent about things outside the 7×7 sub-arena a lot and maybe caused our agent to not get good at chasing after or killing opponents.

This error, the previously mentioned badly chosen rewards and some other things are corrected in a new version. Since this new version led to a much better performing agent, we uploaded the changed code and newly trained model into our GitHub repository at the directory `agent_code/agent2_neural_state2` and will submit it along with this report. It would be great, if you'd use our new agent for the tournament, but since it's two weeks past the official deadline, we at least hope, that you will have a look on it and check how well it plays. The changes made in the state representation are stated in subsection 1.3 and the changed rewards can be found in Table 2 and Table 3. The new code version was used to obtain the results discussed in the following parts of this section.

Important Parameters Given a certain state representation, the four main influences on how good and fast our model learns are the learning rate α and the discount factor γ, see equation (11), as well as the rewards and the exploration-exploitation strategy. Our diminishing ϵ-Greedy strategy uses an exponential decay $\propto \exp\left(-d \cdot n_{\text{episode}}\right)$, where the main parameter is the decay d. In all cases we initialized every model with a previously trained one, which was already trained over 70 000 episodes and did quite well at destroying crates and collecting coins. But since the exploration rate ϵ of the new trainings is still high after 20 000 episodes, the impact of this initial model is probably just a small one.

3.1 Learning Rate α and Discount Factor γ

We used the same rewards from Table 1, except for the five in Table 2, and measured the performance every 1000 episodes for over 100 test runs in different categories, e.g. average coins per game or suicide rate. The results can be seen in Figure 1. We applied a gaussian filter to the data to, because it looked a lot too "noisy" to see the differences between the models. Therefore, the observations about each models performance are only valid for the mean over multiple games, though sometimes one model performs really well but another time equally bad.

The three models, which use $\gamma = 0.95$ but different learning rates $\alpha \in \{0.3, 0.65, 1.0\}$, are performing almost equally good and diverge only a bit in their suicide rate, or survival rate respectively. But obviously the discount factor has a bigger impact, since the fourth model with $\gamma = 0.8$ differs in every category, except for the suicide rate.

INVALID_ACTION	-15	OPPONENT_ELIMINATED	0
COIN_COLLECTED	110	SURVIVED_ROUND	0
KILLED_OPPONENT	500		

Table 2: Changed rewards for training and testing with different α and γ

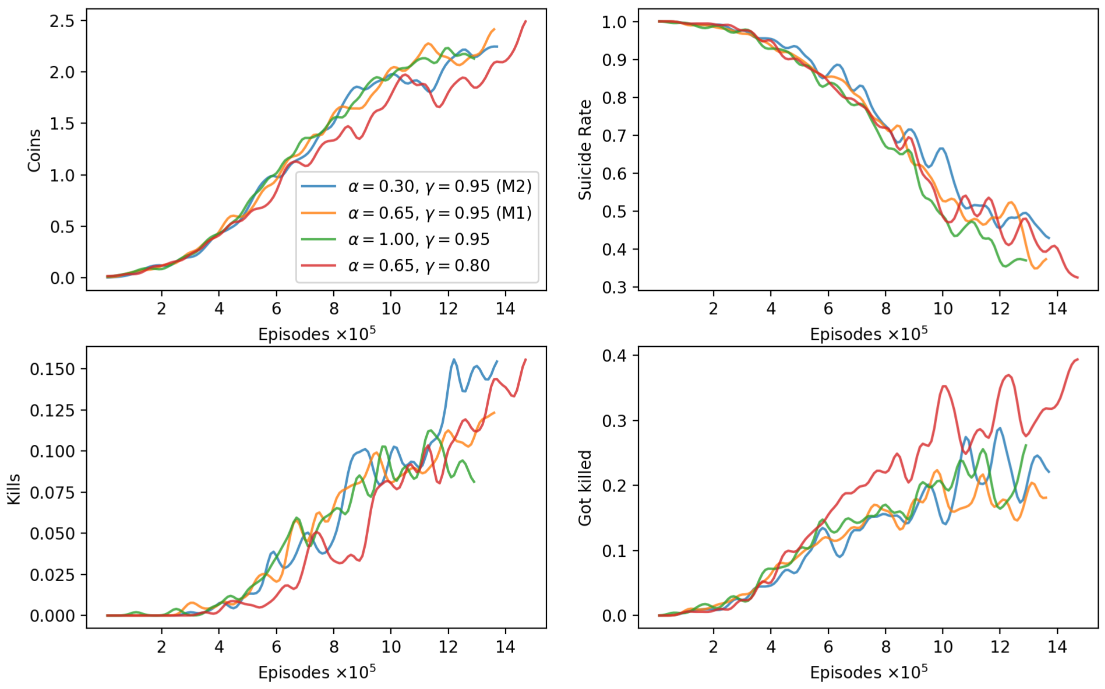

Figure 1: Mean performance of models over 100 test episodes ($\epsilon = 0$) in various categories with different $\gamma \in \{0.8, 0.95\}$ and $\alpha \in \{0.3, 0.65, 1.0\}$, as specified in the first plot

3.2 Diminishing ϵ-Greedy and Rewards

This time we kept the learning rate $\alpha = 0.6$ and discount factor $\gamma = 0.95$ constant, but varied the exponential decay $d \in \{0.00001, 0.000032, 0.00005\}$ of our diminishing ϵ and also tried some other reward combinations, which differ from Table 1 for the events stated in Table 3. R1 is the same as in subsection 3.1 and the other two are slightly different.

The differences are a lot bigger than in subsection 3.1, but only regarding the time needed to learn. Probably, if we trained longer, the models would eventually converge and we could see differences in the final performance.

A problem with every of our agents is, that as soon as all crates are destroyed the agent often stops moving. He also rather moves away from opponents than to actively try to kill them. The former is probably due to the high reward on destroying crates, but we also tried to keep it smaller, which only led to a non convergent model.

Events	R1	R2	R3
BOMB_DROPPED	-2	-5	-2
CRATE_DESTROYED	40	40	40
COIN_COLLECTED	110	110	200
KILLED_OPPONENT	500	700	500
KILLED_SELF	-300	-400	-200
GOT_KILLED	-100	-400	-400
SURVIVED_ROUND	0	400	500

Table 3: Different reward combinations for training and testing our model

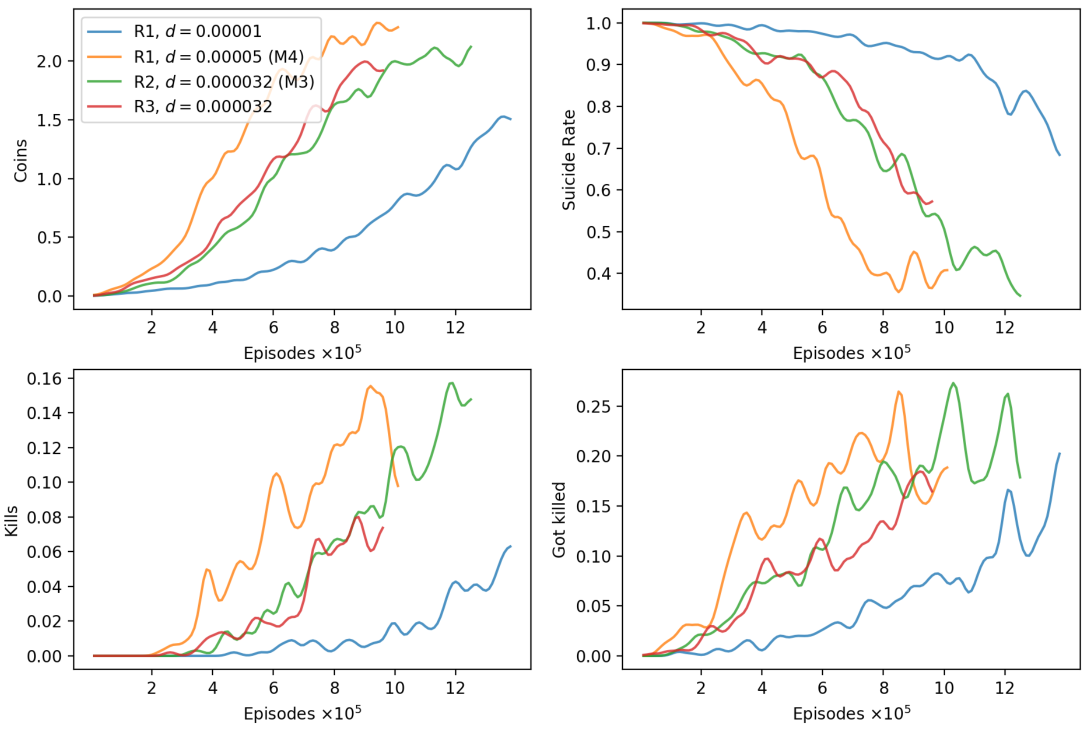

Figure 2: Mean performance of models over 100 test episodes ($\epsilon = 0$) in various categories with different Reward combinations, see Table 3, and ϵ decays d, as specified in the first plot

3.3 Final Evaluation

In this last subsection of our experimental studies, we finally measured how good the models are, whose trainings we discussed in the two previous subsections. We chose the four best agents and let each play against three simple agents, as well as let them play against each other, to find out how many points they can get and how often they'll win.

The parameters and rewards used are stated in Table 4. We let each agent play against three simple agents over the course of 1000 episodes, for which we computed the average score and also how often they won, lost or had a draw with one of the simple agents. The results can be seen in Table 5. We also performed a game of 1000 episodes, where our four agents played against each other. The results are in Table 6.

Model	Rewards	α	γ	d
M1	R1	0.65	0.95	0.000032
M2	R1	0.30	0.95	0.000032
M3	R2	0.65	0.95	0.000032
M4	R1	0.65	0.95	0.000050

Table 4: Rewards, learning rate α, discount factor γ and decay d of exploration measure ϵ for different models used for final comparison

Model	Average Score	Victories	Defeats	Ties
M1	3.059	273	630	97
M2	3.476	230	691	79
M3	3.652	304	613	83
M4	3.135	188	739	73

Table 5: Average Score, Victories, Defeats and Ties of each model when playing against three simple agents for 1000 episodes

Model	Average Score	Victories	Defeats	Ties
M1	2.349	197	698	105
M2	2.580	235	629	136
M3	2.489	206	657	137
M4	2.175	146	724	130

Table 6: Average Score, Vicories, Defeats and Ties of the models when playing against each other for 1000 episodes

The model M3 works really well when playing against simple agents, maybe because of the higher reward for killing an opponent and stronger punishment for getting killed or killing itself. Despite the fast learning of M4 (Figure 2), it doesn't perform so well, as one can see in Table 5 and Table 6. When playing against each other, the best of the four models is M2. But since it doesn't perform as well as M3 when playing against simple agents, we can't make a good guess for why this is.

Finally, we chose to submit model M3 along with this report and also to our GitHub repository in the `agent2_neural_state2` directory. The model was trained in the same way as described in subsection 2.3.

4 Outlook and Ideas

4.1 Our Model

Difficulties As it probably becomes clear throughout our report, we have tried a lot different approaches and didn't arrive at an optimal agent. We had a lot of difficulties finding better state representations, parameters, etc. Though we resolved a lot of these problems, still many uncertainties remain. For example one can see in Figure 1 and Figure 2, that the models didn't converge at all, they still need to be trained for many more episodes and maybe would become quite good.

We also realised, that our general approach to the project was not structured enough, also due to our lack of experience with deep learning. Reading this report, you probably also realize, that we did try a lot of different ideas and in the end didn't have enough time to focus on how to get the most out of the neural network, e.g. using more layers, train parallel on GPUs, etc.

Learning Environment Also, what we didn't try at all is to adjust the difficulty of the agents opponents. Since the random agents most of the time just kill themselves after just a few steps and the simple agent is already quite good at avoiding to be killed, our agent needs a lot of time learn. If we would work on it for some more time, we could implement some sort of self play strategy or other opponent combinations, which maybe would give a more suitable learning environment.

States On the other hand, the state representation contributes a lot to the learning quality and also the time needed to train. This could be refined and would probably lead to a different strategy of the agent, since the information given to him highly influences how he chooses his actions.

Exploration-Exploitation Another big impact on the overall learning and performance of the agent is the exploration-exploitation strategy, which in our case was a simple diminishing ϵ-greedy. One problem here is, that as long as ϵ is large enough, the agent "explores" a lot and thus often kills himself. A idea to resolve this, is to let him play it safe, as long as there is too much danger, and let him explore as soon as the danger of a bomb is small or absent.

Rewards What we would also test a bit more is are the reward combinations and design of auxiliary rewards. For example, our agent is trained too much on destroying crates and finding coins, but struggles to actively chase and kill enemies. This can maybe be improved by designing appropriate rewards. This can also be reached by guiding the agent actively to the enemies by giving rewards for choosing the right to make approaching an opponent more attractive.

4.2 Game Setup

Working with the bomberman framework is quite intuitive and easy to handle. However, we have a couple remarks of which we think they .

In each step (except the first one) the first function which is called from `callback.py` is `reward_update` to receive the games response after the action of the previous step has been executed. That is why one always has to remember the previous action as well as the previous state to associate them with the rewards gained in the current step, because these are actually the rewards evoked from the action in the last step. For us it would be more intuitive to first execute the chosen action of each agent, update the game, receive rewards and then begin the new step.

Furthermore, for us it would have been helpful to be able to decrease the size of the arena for testing reasons and to begin with smaller state spaces until a good approach is found. Setting the `rows` and `cols` in the `settings.py` just threw an error.

Especially when working with neural networks training on GPUs is useful. Therefore the game framework needs to be parallelized to run several games at the same time on different cores. Since we were busy to figure out a good approach for our learning algorithm we haven't had enough time to come up with a solution to make the game run on GPUs (which would have saved us weeks of training on our lousy CPUs :)). Maybe a modified framework for parallel computing or at least some hints in the projects instructions to accomplish that would be helpful, since parallel computing was not necessarily content of the lecture.

At last, a small but nice feature would have been to disable logfiles, without having to modify `agents.py` and `environment.py`, since they can get quite big.

However, most of the aforementioned remarks didn't have a big influence on our work but may help others in the next project.

5 Summary

As already pointed out in previous sections we came up with a working agent which is able to play the bomberman game. Though we tried different regression models we kept focus on using a Deep Q-learning Network to train our agent. We chose this way to handle the large state spaces since implementing common Q-learning with a simple Q-Table and creating an appropriate state representation to keep the state space small didn't seem appealing and elegant to us.

In retrospect we should have invested more time in rewriting the game framework to make it run on GPUs (i.e. train on amazon servers) to speed up training and to be able to make more structured investigations. Otherwise we should have sticked with regression trees or common Q-tables.

Nonetheless, we clearly showed that our DQN approach as well as our chosen state representation is reasonable and lead to a working agent which performs better than the `simple_agent`. Even though we weren't able to really use the whole potential of neural networks, we gained a good insight in how reinforcement learning, especially using Q-learning, can be used with regression models (DQNs, Trees, Linear Regressions,...).

We are convinced that our agents performance could be increased substantially by giving it more training time.