FOML final project - Learning to play bomberman

Jan Gräfje, Lukas Wenzl

March 25, 2019

0 DISCLAIMER

In the course of this report we are going to sometimes attribute active verbs, motivations etc. to agents in the game. This is not to imply "agency" in the strong sense of the word on the part of the game agent, but merely used to provide an intuitive understanding of the behavior of the learning algorithm in anthropomorphized terms.

1 Technical requirements & structure

The code of our project can be found in the following public github repository:
https://github.com/lukaswenzl/bomberman_rl

The repository contains 5 subdirectories, the final version we would like to hand in can be found in 'hand_in'. The other directories only contain an unmodified version of the code provided with the project instructions ('template') and subdirectories for the three tasks laid out in the project instructions ('1', '2' and '3'). However they are not cleaned up. They are just there to show proof of all our intermediate steps.

We need the following packages (with dependencies) additional to the standard ones:

- Keras
- TensorFlow

2 Overview of the plan [Jan Gräfje & Lukas Wenzl]

We pursued a bottom-up approach. We did not go online and copy a close to finish solution from a paper but instead worked up from what we had from the lecture. This probably leads to us having less impressive solutions than some of the other groups however we actually came up with our own approach and learned a lot along the way. This report will document our process. We tried to solve the problem step by step. We first developed a reinforcement learning code that can find the coins in a game without crates (step 1). We than adapted this code to instead use a simple neural network to solve the same problem. From that we went to finding the coins when they are hidden below crates (step 2). The last step is to also put opponents on the field (step 3). For this we use our code from step 2 that is trained on a field without opponents but only needs minor adjustments. Since the agent at this point can put bombs, run away from them and collects coins.

Due to lack of time, we were not able to progress to step 3 as outlined above and only trained our agent in an arena without opponents. However the outline above represents the plan we had in the early stages of the project.

3 Design + training strategy + results for Step 1

3.1 with reinforced learning [Lukas Wenzl]

As a first step we take a striped down version of the game and try to implement the simplest possible approach to learn a very easy problem. In this case we remove all the crates and the opponents. So the only task will be to pick up all the coins. We simplify this even further by introducing an extremely strong feature. This feature that describes the state s of the game will only have 4 possible states. The feature is just which direction will lead to the closest approach to a coin. The encoding is the following: 0 for "UP", 1 for "RIGHT", 2 for "DOWN" and 3 for "LEFT". We will start with a random Q-matrix and will then update it after each game. Ideally the Q-matrix should approach a unity matrix. This is a simple test to see if our algorithm structure is working and the method works in principle. Once this is working correctly we can increase the difficulty step by step to reach the full complexity of the final game. We implement everything inside of our agents code. The approach follows the following structure (partly pseudo code):

```
in setup:
    self.experience  = []# will contain training
                        #data with [[s,a,r],[s,a,r],...]
    self.q_matrix = np.random.rand(4,4) #random Initializing
    self.hyperpar = {"y":0.4, "eps": 0.5, "lr":0.2, "training_decay":0.99,
    "mini_batch_size":100}    #y, eps, learning rate, decay factor alpha

in call:
    eps *= training decay
    if random() < eps or np.sum(self.q_matrix[state, :]) == 0:
        self.next_action = np.random.choice(['RIGHT', 'LEFT', 'UP', 'DOWN'])
    else:
        a = np.argmax(self.q_matrix[state, :])
        self.next_action = number_to_actions[a]
    reward = 0
    if(state != actions_to_number[self.next_action]):
        reward = -0.1 #punish deviating from best path
    self.experience.append([state, actions_to_number[self.next_action], reward])

in reward update:
    if(COIN_COLLECTED):
        reward = 1

in end of episode
    for i in np.random.randint(len(self.experience)-1, size=mini_batch_size):
        s, a, r = experience[i]
        new_s, new_a, new_r = experience[i+1]
        q_matrix[s, a] += r + lr * (y * np.max(q_matrix[new_s, :]) - q_matrix[s, a])
```

A short summary of what this pseudo-code does:

- In 'setup' we create all the things we need: an array for experience, set hyper-parameters and create the Q-matrix with random values between 0 and 1.
- In 'call' we use the Q-matrix to decide where to go. With probability epsilon however a random direction is chosen. This epsilon decreases over time.
- In 'reward update' the agent gets reward 1 for finding a coin and reward -0.1 if he goes off the optimal path to the next coin.
- After each game (in the function 'end of episode') the Q-matrix gets updated with a certain number of random training instances (mini_batch_size). To update the Q-matrix

a 1-step Q learning approach is used. So overall this implementation forms a 1-step Q learning approach with ϵ-greedy action selection for exploration.

As expected this converges quite fast. The ideal solution should be a Q-matrix where the diagonal elements are far bigger than the other elements, since that means that the algorithm always follows the optimal (precomputed) path. We use the sum of the diagonal elements minus the sum of the off diagonal elements to measure the convergence. This should increase with time and at some point level off. This can indeed be observed in Fig. 3.1. We however also observe that this convergence is not very stable. Without giving negative rewards or increasing the learning rate by too much stops the algorithm from working. Then it does not converge anymore. This indicates to us that we must select our hyper-parameters and rewards very carefully.

We can also let the trained program run in graphics mode and see that the algorithm now runs from coin to coin and collects them as expected.

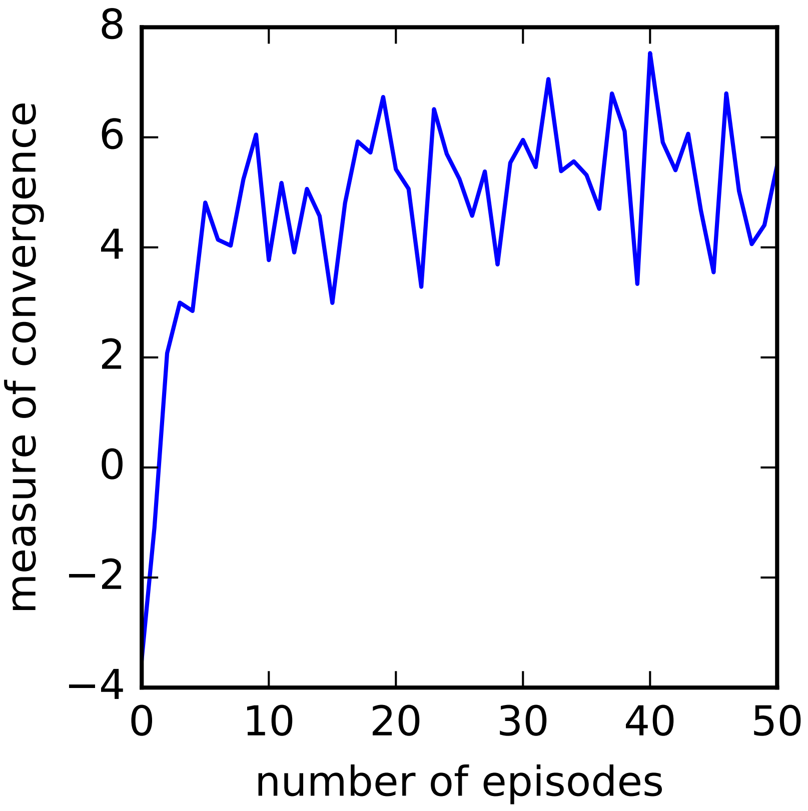


Figure 3.1: Number of episodes vs the measure of how much the diagonal elements become dominant. This shows that our simple test of reinforcement learning does converge.

3.2 WITH NEURAL NETWORKS [LUKAS WENZL]

Instead of the learning of a Q matrix we can also use a neuronal network. These have the advantage that we do not have to decide on the features by ourselves, but rather the algorithm figures them out by itself. In theory we can just put all information about the game-state, that we have, as input and as an output the next action. However we do not go straight to this very general implementation but rather try some intermediate steps first.

Here we adopt a simple neural network with 4 input nodes and 4 output nodes. The output nodes are chosen with the same encoding as before. If the first one is the highest we choose the action up, for the second right, for the third down and for the fourth left. And for the input we try a few different things to learn how well the training of the neural network works. For the hyper-parameters we adopt the same system as before. Just now instead of taking the Q matrix to choose the next step we use the prediction from our neural network. The pseudo code looks like this now:

```
in call:
    eps =eps*training_decay
    if np.random.random() < eps:
        next_action = np.random.choice(['RIGHT', 'LEFT', 'UP', 'DOWN'])
    else:
        a = np.argmax(model.predict(np.identity(4)[state:state + 1]))
        next_action = number_to_actions[a]

rewards:
    if(did not walk towards next coin):
        reward = -0.01
    if(coin collected):
        reward = 1

in end of episode:
    for i in np.random.randint(len(experience)-1, size=mini_batch):
        s, a, r = experience[i]
        new_s, new_a, new_r = experience[i+1]

        target = r + y * np.max(self.model.predict(new_s))
        target_vec = model.predict(s)[0]
        target_vec[a] = target
        model.fit(s, target_vec.reshape(-1, 4), epochs=1, verbose=0)
```

Again at the beginning we mostly do random choices (high eps) and over time the algorithm trains (model.fit(...)) and takes over the action.

The easiest thing to try is to take the same input as we had above (step1_v2). This was basically giving the answer to the algorithm and seeing if it realizes that it just has to give it back. So in this first test of the neural network we just gave the path to the closest coin. As a reward we only use the collection of the coin. In Figure 3.2 we plot the number of games trained vs the amounts of coins collected.

This clearly shows that the implementation is working on some level. With increasing number of games the agent on average collects more coins, which is the result we desire. Again the solution is not very stable. As a next step we try to give the coordinates of the agent and the coordinates of a coin as an input (step1_v3). This problem is quite a bit harder to solve. To make it converge we introduce a new soft penalty. Every time the agent does not walk towards the coin we put as an input we give a penalty. The convergence is also visualized in Figure 3.2. This takes a lot longer to converge. For every position the agent can be it needs to be trained with all possible locations that coins could be in to learn where to go each time. One possible way to improve this might be to center the coordinate system on the agent (step1_v4). This way a coin that is 1 step to the right relative to the agent is the same problem to solve where ever it happens on the game board. So the idea is that the convergence will be much quicker. Indeed we can see in Figure 3.2 that step1_v4 converges much faster than step1_v3 and it also seems to be more stable. Interestingly step1_v4 is as fast in training as step_v2. The approach where we do not center the coordinate system (step1_v3) takes a lot longer to converge. This is to be expected since it has to learn about a lot more scenarios before it can work effectively. We do not directly dictate the answer anymore so we are coming closer to the true goal of having the agent figure out how to play basically by itself with only weak penalties and rewards. All three approaches have the issue that sometimes the agent gets stuck. Maybe it will be necessary to make some kind of loop detection similar to what the 'simple agent' provided with the project assignment has.

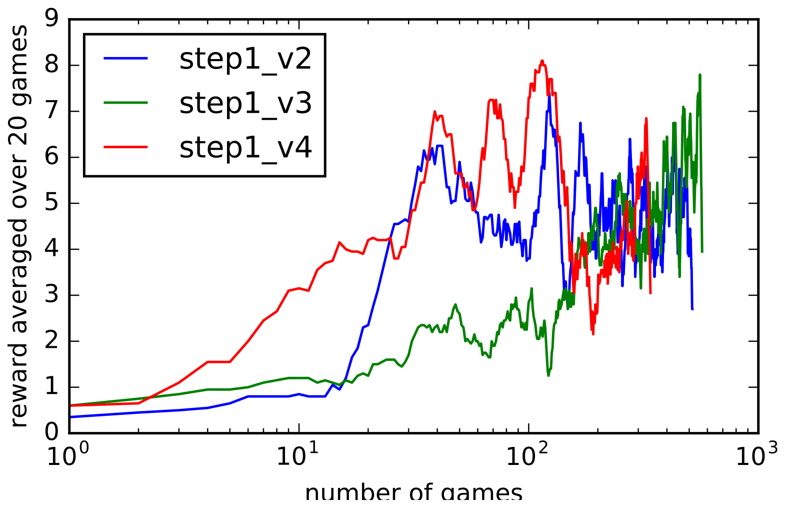


Figure 3.2: Number of episodes vs average number of coins collected for the different inputs into the neural network.

4 Design + training strategy + results for Step 2

First it is necessary to adapt the game to the requirements of the second task. However this can easily be achieved by disabling all players besides our agent in main.py. Also we adapt the random actions the agent can take in such a way that it also drops bombs sometimes.

To complete the second task, namely dropping bombs to find hidden coins and to collect these coins subsequently, the agent has to learn several concepts:

- Dropping bombs destroys crates in the explosion zone.
- Destroying crates has the potential of uncovering coins.
- Destroying crates also increases the space that can be explored by the agent.
- Dropping bombs can also destroy the agent, which is to be avoided at all costs.

4.1 Experimentation with rewards [Jan Gräfje]

Since the task for the agent is way more complex than the first one, design of rewards will have to be more nuanced than before. If we were to use the same rewards as before (i.e. positive reward for collecting coin, small negative reward for each turn the agent takes) this leads to an undesirable outcome; at the beginning the agent sees no coins and has no motivation to destroy crates (since there is no direct reward for this). However he has a motivation to kill himself, because this reduces the amount of turns per game and therefore the incurred negative reward.

It is quite clear that intermediate, "soft" rewards have to be established. Events that we generally consider to be positive are

- Agent survives the round (i.e. he did not kill himself)
- A coin is found (since we train the agent alone in this step, all found coins are found by him)
- Agent collects a coin
- Agent destroys a crate or even several crates

while we consider the following to be negative

- Agent kills himself
- Agent takes an invalid action (should hardly be relevant, since there are filtered out beforehand)

To begin with the same implementation as in the first task was used and only the distribution of rewards was adapted, because we wanted to experiment with the effect of different reward strategies at a very basic level.

In the first experiment we placed a high reward on the agent surviving the round, since this seemed to be an obvious prerequisite for long-term success (at least for humans survival is almost always the highest priority). But after very short training it converged to just moving up and down in the corner of the arena without ever placing bombs (and thereby surviving consistently). This is a dead-end for training, but a lesson can be learned from this: Rewards have to be designed in a way, such that they there is no "easy" way to collect a reward without working towards the goal of the game, since that effectively prevents exploration of possible game strategies.

Accordingly the reward for surviving was set to 0 in following experiments (while there was still a negative reward for the agent killing himself) and heavier emphasis was placed on exploring the arena (by increasing the reward for the destruction of crates and finding coins). However there was still a tendency for the agent to become stuck in an oscillating pattern in the corner of the arena. To discourage this we again introduce a small negative reward for every turn taken; an alternative would be to assign a negative reward if the agent repeats the movement multiple times. This resulted in a small improvement after some training, but more fundamental changes to the way the network is trained seem to be necessary.

4.2 Encoding game environment [Jan Gräfje]

Overall the agent seems to require much more information about the game environment than in the first task. Especially since he has to plan several moves into the future to avoid being blown up by his own bombs. This motivated us to pass a lot more information to the agent than in the first task.

The easiest approach to achieve that appeared to be to simply pass the complete arena to the agent as input for the network (stored in self.arena and containing the locations of walls, free tiles and crates). Additionally we encoded other desired information onto a copy of the self.arena object so that we arrived at the following encoding:

-1	wall
0	free tile
1	crate
2	coin
3	bomb
4	other agent
5	agent itself

There is one concern with encoding information in this way: The network could come to the conclusion that there is some information in the ordering of the integers (e.g. "High integers are bad, low ones are good"), which is not intended by our design and which could restrict the possible learning outcomes in an undesired way. Online research revealed this to be a common problem with ML implementations, but it also presented "one hot encoding" as a solution to this problem. In this procedure our integer encoding is effectively transformed to a binary representation by sparse 1D arrays. While it appears plausible that this avoids the aforementioned problem, it comes with the drawback of significantly increasing the size of the input to the network from a 17x17 array representing the arena to a 17x17x7 array in which each of the possible elements from the table above would be represented in its own array dimension.

We are afraid that this would be so detrimental to training performance that we do not implement it at this early stage, in which we are still experimenting with the network a lot, but we keep it in mind for potential future implementation when we are more satisfied with other aspects of implementation. Also principal component analysis (PCA) could potentially be a worthwhile approach to this problem, since it could help the network distinguish which information from the (largely sparse) 17x17x7 array is actually relevant for playing the game.

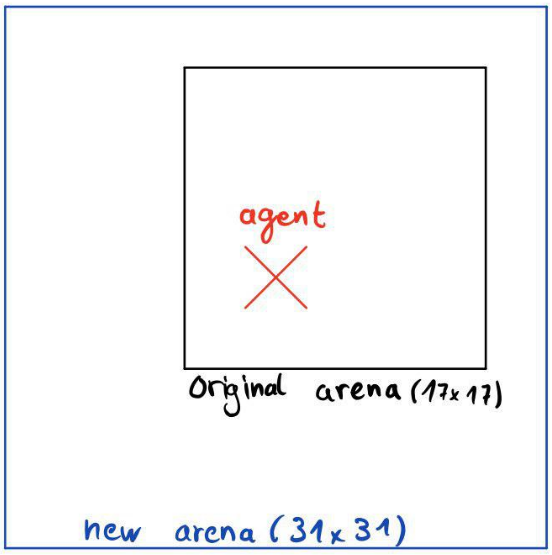


Figure 4.1: Visualization of arena projection

Another problem is that the agent is not always starting the game in the same corner of the arena, which poses an additional difficulty for the network to figure out. To avoid this we project the 17x17 arena into a larger arena in such a way that the agent is always in the central field and provide this new arena as input to the arena. Therefore the input at the center of the arena always has the highest relevance, irrespective of starting position. To fit the whole arena into the new arena it has to have relatively big "margins" and in each turn a large part of the new arena is unused (since there is no relevant information projected to these parts of the arena); the unused elements are set to "-1" since the agent is not actually able to enter them.

However training the network on this implementation for one night did not yield any recognizable learning progress. Our conclusion is that the algorithm takes too long to discern which information from the large array is acutally relevant. Therefore our next step is to only pass an arena of reduced size to the network (but still centered on the agent). A 9x9 input seems to be the smallest reasonable input, since this way an agent can see a placed bomb before stepping into the explosion zone.

This way of course we loose a lot of information. So in the next step we will use PCA analysis to reduce the input to a reasonable size.

4.3 Reducing the input dimension with PCA [Lukas Wenzl]

The game board is quite large and by centering it, we make it even larger. Furthermore by using one hot encoding it gets larger by another factor of 7. To reduce the input dimension without loosing a lot of information it is a good idea to perform dimensionality reduction. We will do this with PCA, a method we know well from the lecture. To implement this, we need a training sample since it is a supervised machine learning method. This training sample has to have a bunch of examples of how the game arena might look like in different stages of the game. Luckily we have the simple agent available. We can just run the simple agent 50 times and save the different game situations it encounters. We then use this to train a PCA. To emphasize the close proximity of the agent we do this twice: once for the whole field and once for a square of 9x9 around the agent as discussed in the previous chapter. We then fit an instance of sklearn's PCA implementation to the data. We can then save this instance using pickle and reload it with our agent. Then we can reduce the size of our input by transforming the full game board with the instance of PCA. We choose the most prominent 30 components for the small and complete board. This alone did not drastically reduce the training time after some quick testing by letting it train for some time. Clearly it did still not learn how to use a bomb. So we decided against using the PCA for the next steps. Our assumption is that not enough information is contained in the few components we choose.

4.4 Exploration strategy [Lukas Wenzl]

We also had the idea that instead exploring with random moves we could explore with the moves the simple agent would make. This can be implemented easily in our code. In the first few games with a certain chance a random move is chooses. Instead of the random move we now call a method in which we copied the code of the simple agent. This is of course only ever used i training mode. So after some time and outside of the training only the moves suggested by the neural network are used. Only to give the agent a head start we use the simple agent logic. We implemented this on top of the PCA approach from the last paragraph. As we already mentioned this might have doomed it to fail already however we also did not see any kind of improvement in terms of learning. After a few hundred games the agent still blew itself up or did nothing. Either 0 or 1 coin was collected, games typically ended within 50 steps because of a self kill. So we also decided to not go forward with this approach either.

4.5 Reducing training time [Lukas Wenzl]

To more effectively train in order to see if our input and neural network structure is in principle able to solve the task we will use a few methods to increase the speed of the training. This is necessary since our computing resources are limited and we can't retrain the agent many times during the time of our project. One part of the programs is very easy to help the agent with to get a jump start: tell him how to run away from an active bomb. The logic to hard code this is very simple. When we are standing on an active bomb we simply suggest to the agent to move away from it. The main goal for the project is for the agent to develop its own tactic in terms of how to explore the field, how to attack enemies and how to collect coins. To get to this step faster we help the agent out a little in avoiding getting blown up after the first few steps in most of the first few hundred games. We do this by adapting the code from the simple agent. If the agent is standing within a bomb radius the code will check if the decision by the neural network is a valid step to leave the bomb radius if not it will instead perform a valid step away from the center of the bomb. Additionally if the agent tries to drop a bomb at an inappropriate location we instead choose a random different valid action. We only allow dropping a bomb in a dead end or within 2 blocks of another player. This is a very effective approach. Instead of holding still or blowing himself up the agent now right away starts to explore the game field.

4.6 Giving the neural network more information [Lukas Wenzl]

Currently our input is a 9x9 array of the immediate surroundings of the agent. During testing with the GUI turned on we observed that coins that are too far away were ignored by the agent which makes sense. Once a coin is more than 4 fields away the input does not contain any information about it. Furthermore the agent often tries to perform invalid actions which slows him down. There is multiple ways to avoid these issues. We decided to add an extra row to the input. So 9 extra nodes. We add these additional nodes to give the agent more information about the game situation. 5 nodes to encode whether the actions wait, left, right, up, down, bomb are valid (1) or not (0) and 4 nodes with the relative distance in x and y direction two the closest two targets. Targets are agents, dead ends and other players. This is similar to what we did for step one and there the agent was able to find the coins with the coordinate information alone so it should help the agent to make an informed decision. With this we arrive at the final

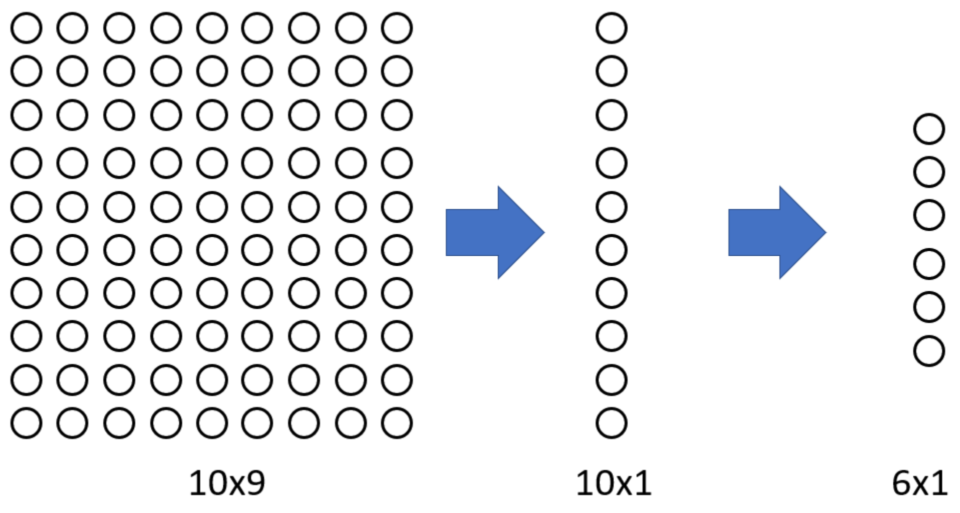


Figure 4.2: Schematic showing our final neural network structure. We have a 10x9 grid as input, a 10x1 hidden layer and 6 output neurons that encode which action to take in the next step.

structure of our neural network. We use a grid of 10x9 as an input, which is flattened before handing it to the next layer. This input contains the 9x9 field around the agent, a one hot encoding of the valid actions and the coordinates of the closest two targets. Fig. 4.2 visualizes this input structure. It also shows the further structure of the neural network. There is one hidden layer with 10 nodes. The agent can use this hidden layer to make conditional decisions more effectively. In our previous experiments more hidden layers where unfavourable so we keep only one hidden layer. The last layer has 6 nodes and forms the output. Each node is for one action and we choose the one with the largest value.

4.7 Handling long training [Lukas Wenzl]

So far we stored every step the agent makes. For this we store the input to the neural network, the reward received and the move that was performed. After a few hundred games this gets to large. Also we sample from this array for the training. More recent game moves are more relevant to the training than older ones. So to address this and reduce the size of the array we only store the last 10 000 moves. Everything older than that we delete. This automatically prioritizes the more recent moves from approximately the last 30 games.

4.8 Loop-breaking [Jan Gräfje]

Training the network for some time and then looking at a few games with the GUI enabled showed that a major problem for the performance for the agent is its tendency to get caught in loops. This prevents the agent from further exploring the arena and finding all the coins.

To prevent this, we implement an explicit loop-breaking function: If the agent has been at the current spot four times in the last 20 turns, he is to take a random action that has to be different from the action he took two turns before (e.g. if the agent is alternately going up and down, he is forbidden from going up while being in the lower position). This is a very explicit solution and as such not very elegant or applicable for a wide range of problems. More complex loops (e.g. running in circles) will not be forbidden by this, since in these cases it would be necessary to forbid an action that was taken at a time further in the past. However we have come to the conclusion that two-step-loops are by far the biggest issue and that we accept the shortcomings of this fix in favour of its simplicity.

4.9 Batch Normalization [Jan Gräfje]

One other common with training neural networks is that inputs (or values in the hidden layers of the network) may not be normalized or centered in a way that aids fast training of the network. One convenient way to handle this is Batch normalization. It is basically an additional transformation we add to every layer of the network, that normalizes the data before putting them into the next layer. Since this is already implemented in Keras very efficiently and easy to add to our code, we also include it in our work. We were not able to observe any direct consequence of this. However we are quite sure, that it does not slow down our code significantly or introduce any unwanted other effect, so we keep it in the code in the hope that it at least speeds up training somewhat.

4.10 Neural network architecture [Jan Gräfje]

Since we increased the size of the input into the neural network significantly, it seems reasonable that the size of the layers of the neural network should also be adapted. Up to now we used a single dense layer with 10 neurons. We now experiment with a network with two layers, where the first layer has as many neurons as the input array has elements and the second has a smaller size of again 10 neurons.

To understand how this affects the training we let this network train for a night on the first task and compared it with the performance of the previous network (see Fig. 4.3):

Although the 2-layer network was trained on a ten times higher number of games than the 1-layer network, it performance was still significantly worse. It is possible that the 2-layer network would outperform the other after a much longer training time; however we do not have hardware to utilize this. Therefore we abandon this avenue of experimentation.

It is also imaginable that there are other network architectures that would be more successful. However at this time we have no clear idea, how these could look like, and no time for an in-depth exploration of literature.

4.11 Parameters for training and results of training [Jan Gräfje]

After previous experimentation with reward strategies, we decided to hand out rewards in the following way for the final training of the agent:

destroying a crate	1 per destroyed crate
finding a coin	15
collecting a coin	40
killing itself	-5
surviving the episode (to round 400)	0
trying to take an invalid action	-10

The reasoning for this reward strategy is as follows:

- Rewarding the destruction of crates is very important to motivate the agent towards exploration in the early phases of training. The reward is handed out on a per-crate-basis since a higher number of crates destroyed increases the probability of finding a coin.
- Rewarding the finding of a coin further rewards exploration of the arena. Furthermore it especially rewards destroying crates in areas where coins can actually be found (e.g. in areas of the arena where up to this point less coins have already been found).
- Collecting a coin should obviously be rewarded, since this is the desired outcome of training.
- In general the agent should be rewarded for not killing himself with his own bombs. However as mentioned before, rewarding survival actually discourages exploration and rather encourages the agent not to place bombs at all and just sit in a corner. Furthermore the implementation of the hard-coded bomb avoidance has reduced the

need for actual learning in this regard and therefore the rewards should rather focus on the collecting of coins.

Finally we only had a few hours to train the network in this configuration. Therefore it is not that surprising that the result is pretty bad. As is clearly visible in Fig. 4.4 the performance of the agent does not improve with time; rather it gets worse after the first few hundred rounds in which the behavior of the agent is still dominated by randomly chosen actions. Looking at some games in the GUI reveals that the agent still gets caught in loops most of the time and therefore collects less coins with progressing training (when the influence of random actions, that have the potential to break loops, diminishes). We can hope that this problem would work itself out with more training; however this will have to remain speculation, since we have now run out of time to let this train further before the deadline.

5 Applying the code to step 3

When training our code from step 2 with other agents enabled it struggles even more than it already does. We instead decided to train without other agents enabled. We do treat the positions of other agents like dead ends for the input. So the trained agent from step 2 also goes after them. Furthermore the strategy to run away from bombs is independent of who dropped the bomb so this also carries over. This is not an ideal solution however with how far we have gotten in this project we believe is is the most effective strategy. Our agent will start to hunt for coins and once the game reaches the player vs player phase it has the necessary skills to stay alive and also will drop bombs against other opponents. We test this by taking the trained model from step 2 and letting it play against 3 simple agents. Our agent is usually slower to explore than the simple agents so it collects fewer coins and after that in the fight between them it seems relatively random who wins in the end.

6 Conclusion [Jan Gräfje & Lukas Wenzl]

We did develop a functioning neural network that can draw information of the input we construct from the game state and make a decision of the next step. We found good solutions for step 1 where we created agents that use either neural networks or reinforced learning to find the coins in an empty field. This works well enough that the agent finds most coins in most games. We went from almost directly using the best direction as input to giving abstract coordinates to a neural network that then has to interpret the information and decide on the direction it needs to go to. We clearly see that the agent is getting better over time, meaning that we successfully implemented a neural network. Based on our experience from step 1 we developed an approach for step 2. This is much harder since now the agent needs to use bombs. He needs to decide when to drop them and how to avoid dying. Due to our limited computational resources we determined that we help the agent to learn how to avoid bombs faster and reliably. This way the learning process focuses on the actual interesting parts: the tactic for exploring the game field. A lot of adaptations of our first code where necessary in order to achieve reasonable success rates. We extended and pre-processed the input, implemented loop detections and reduced the training time with different methods. In the end our agent was able to move across the game field by destroying crates. He was also able to pick up some coins. Within 3 hours of training he was not quite able to find the coins as effectively as the simple agent who is always using close to the ideal path. However we do see an improvement with training time and clearly the agent picked up some skills necessary to solve task 2.

7 Possible improvements to our code [Jan Gräfje & Lukas Wenzl]

The first improvement we would make with more time is letting the program train longer. We went through many design iterations and we just could not let it train for a day each time to clearly see how well it works. Especially for our final agent an extra day of training would have improved it. In the end we were only able to train it for a few hours and it became quite clear that this amount of training is insufficient for a task of this complexity.

We should also further think about the input. Our quick attempt at dimensionality reduction with PCA was not very successful however in general it is definitely desirable to pre-process the input in a way to reduce the amount of neurons. Especially if we go back to using one hot encoding which multiplies the amount of input. In general we should use convolution for this issue. For example there is the concept of convolutional neural networks. These are especially adapted for such cases where the input gets very large. It is commonly used in image processing, because it works particularly well, if inputs that are 'close together' also have a close relationship to each other. This of course also applies to this task, since elements that are closer to our agent are in general much more important than those on the other side of the field. However exploring their use was not possible for us because of time constraints.

Another improvement we would make with more time is the training strategy. Instead of starting the training from a blank network each time we could first train the network for an easy problem like step 1 and then use the weights of the trained model as a prior for step 2 and so on. The idea here is that training the neural network on going after coins is easier with no obstacles. If there is obstacles the algorithm needs many more tries to encounter the same amount of instances where it has to go after a coin. If that logic is already contained in the prior this should go much quicker.

And finally one should also probably reconsider the fundamental strategy of building the model part by part and in each step experimenting with different network architectures, hyper-parameters etc. Because during the course of this project we noticed that it is really hard and time-consuming to do this, since after each change the network should be retrained to examine the effect the change had. But if, as in our case, each retraining takes $\mathcal{O}(day)$ this is simply a very time-consuming and frustrating endeavor. Furthermore a lot of components actually already have to be implemented in a useful way for the training to have any chance of actually yielding an at least somewhat capable agent, meaning that the part-by-part approach is facing big obstacles from the start. Therefore one should have probably spent more time for a comprehensive research into available literature and capabilities of ML-libraries in the beginning of the project and then adopted a general approach and architecture found in this time. Then there would have been more time in the end to fine-tune and train this implementation for our needs.

8 Project improvement suggestions [Jan Gräfje & Lukas Wenzl

8.1 Lukas Wenzl

All relevant techniques for the final project were either covered during the exam period and very rushed or not covered at all during class. This way the learning curve was extreme. We spent many hours reading up to only getting a very basic neural network to run since we have no previous experience with it. If this extra work of having to read up a lot to get anything running at all gets taken into account during the grading it is fine, otherwise it is a quite harsh disadvantage.

Also in general a research project with some of the techniques from the lecture would have been much more relevant to our scientific career. So overall we would have preferred the old concept of a research project. I really enjoyed the homework assignments. I think I learned a lot from them. So a final project closer similar to them would have likely also been more enjoyable.

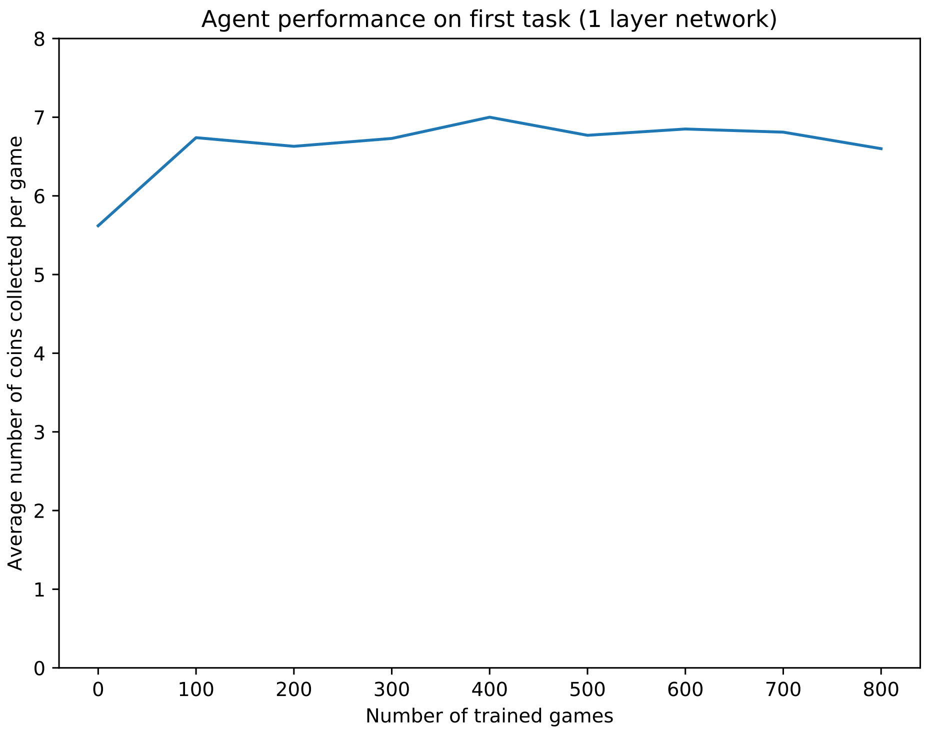
8.2 Jan Gräfje

During the lecture I actually found the approach of the exercise sheets quite useful, in that they provided a structured and gradual introduction to the topics of the lecture. In that light I found it to be quite a shame, that we were for the most part not able to use the actual techniques we learned during the semester in the final project, but rather had to start from scratch in many aspects.

It would of course also have been possible to use a reinforcement learning (RL) approach for the project, which we also covered in the lecture. However when first contemplated this at the outset of the project, we also noticed that (at least from our perspective) there were still some intricacies and details of implementation of RL that were not clear to us after the final lecture (since there was also no exercise covering this topic), which is also why decided not to pursue this. On the positive side it was of course an opportunity to learn some things about neural networks that went further than what we did in the lecture.

However I think I would have profited more, from doing some project utilizing the classification techniques we worked with in the lecture a lot, since then I would not have needed to spend time reading about a lot of new things but rather would have been able to deepen my previous knowledge and get to a more interesting level there.

And while I understand, that a tournament of relatively easily comparable game agents is way easier to handle for you in terms of correcting than individual projects, I think that I would have enjoyed a project of an individually chosen topic more.



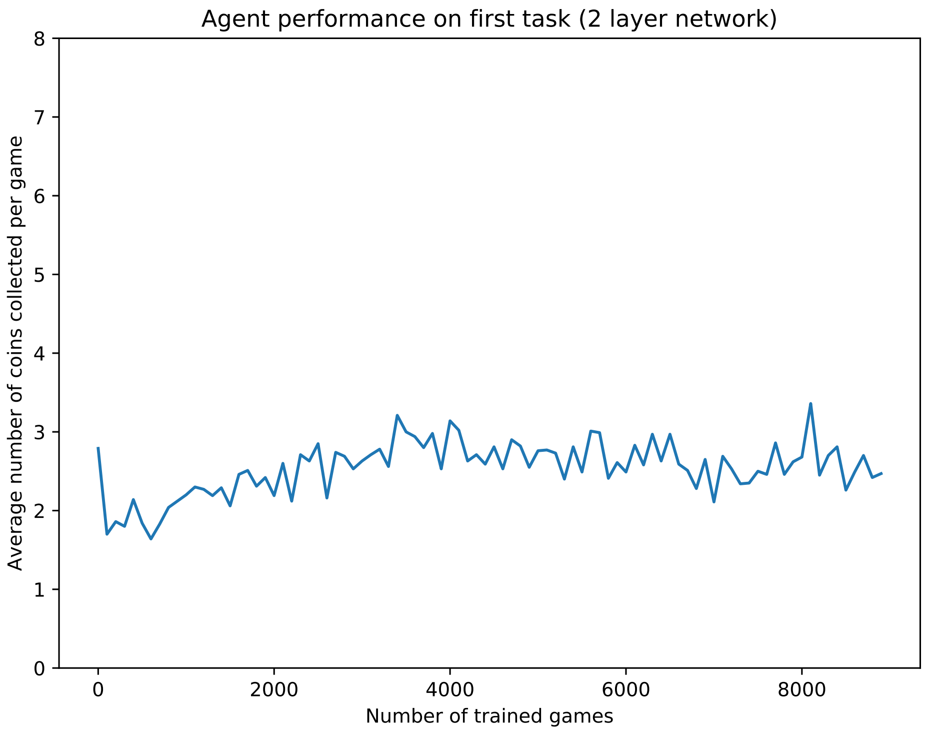


Figure 4.3: Comparison of 1- and 2-layer networks

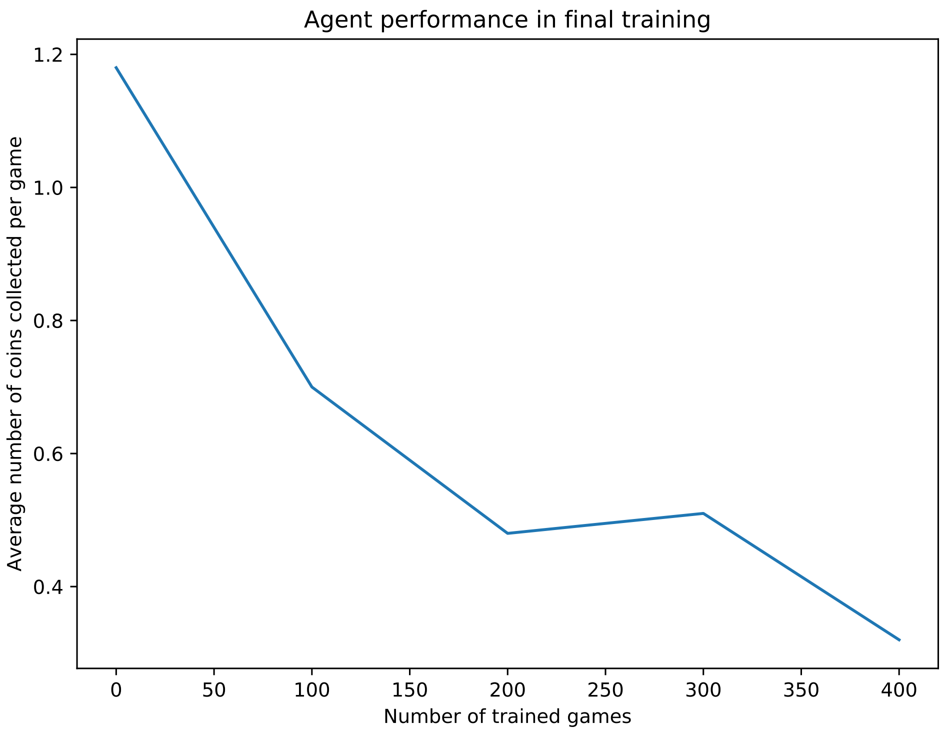


Figure 4.4: Comparison of 1- and 2-layer networks