Project Report for Fundamentals of Machine Learning

Authors:

David Scheid and Patrick Günther

Submission Date: 29.4.2021

Contents

List of Figures

1 Reinforcement Learning (Patrick)

In this chapter, we very briefly discuss the underlying reinforcement learning approach we used for training our agent and summarize our overall training process, as well as the different models we explored during development and their input features. We then explain how we chose hyperparameters for these models.

1.1 Q-Learning (Patrick)

As a general framework of our approach, we chose Q-Learning [WD92]. The goal of this is to maximize a discounted, cumulative reward

$$R_{t_0} = \sum_{t=t_0}^{\infty} \gamma^{t-t_0} r_t, \tag{1.1}$$

also called the *return*, which depends on a policy π. Here, γ represents a discount factor between 0 and 1 to make the sum converge. For each state s, a value function

$$V^{\pi}(s) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t \cdot R_t | S_0 = s] \tag{1.2}$$

describes the expected return. Q-Learning introduces the idea, that if a function $Q(s,a)$, that computes the expected return from a state s and an action a, can be learned, the policy, that maximizes the return can be deduced from that [EM04; Py17]:

$$\pi^*(s) = \arg\max_a Q^*(s,a). \tag{1.3}$$

$Q^{\pi}(s,a)$ is also called an action value function and is defined by [Fa20]

$$Q^{\pi}(s,a) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t \cdot R_t | S_0 = s, A_0 = a] = r(s,a) + \gamma \cdot \mathbb{E}[V^{\pi}(S') | S' \sim P(\cdot|s,a)], \tag{1.4}$$

where $r(s,a) = \int rR(dr|s,a)$ is the expected return at state s given the action a. If we take the right side of the equation and subtract $Q(s,a)$, we get the temporal difference error δ. The goal of the learning approach is to minimize this error. In our case, we use the Huber loss to calculate this error over a batch of transitions, that we saved in a memory [Py17].

1.2 Learning Process (Patrick)

Throughout the training process, we used two models: the target model and the policy model. These have the same structure and are initialized equally at the start of each training. To record game events, we used a replay memory, which can be seen as a queue that records game events, including the last action taken by the agent, the current action, rewards for the transition, as well as last and current game states, and a flag that indicates, if the transition happens at the end of the episode, since no future rewards have to be taken into account in this case.

For every game step, an entry is added to the replay memory. The action is either chosen at random, or by the agent. To define a threshold of how likely the action is chosen at random, we implemented the ϵ-greedy policy, which means that the threshold for each step at which a random action is taken decays throughout iterations. At the beginning of the training we aimed for a decay throughout each episode, which means that the agent is more likely to take random steps at the beginning of the episode and more likely to use the decisions of the model at the end. This is done to encourage exploration.

If the replay memory holds enough entries to create a batch for optimization, we start optimizing the policy model in every game step. This is done by sampling a sequence of transitions from the replay memory. Throughout optimization, the policy model predicts $Q(s_t)$ from which the actions can be computed which should be taken, according to the policy model. The expected $V(s_{t+1})$ values are then predicted by the target model, from which the expected Q values can be computed. The weights of the target model remain unchanged throughout a defined number of steps. The target model is updated then, by duplicating the weights of the policy model. The reason for this is, to stabilize the learning process. We experienced a trade off between updating often and risking big changes in model weights, which can lead to the model not being able to learn, and updating seldom and thereby slowing down the learning process significantly.

1.3 Design of Input Features (Patrick)

We assumed that convolutional neural network would be powerful models for the task, since we could use game states as matrices representing the 17×17 game arena. The problem with one matrix as input for the model would be, that representing different entities would be difficult and could lead to numerical imbalances, if we used different numbers for different entities (higher numbers could have higher impact on the result, even though they represent an equally important entity). For this reason, we decided to create multiple channels for different game entities, resulting in a $6 \times 17 \times 17$ feature tensor.

In general, all fields that are free of entities are filled with zeros. The first layer consists of the overall field, including walls and obstacles. The representation of this layer is the same as the *field* matrix, which is passed by the game state. In the second layer the current position of the agent is modelled, by setting its position to 1, if it is able to place bombs and -1 if it can not. The same encoding is done in layer three, which represents adversarial agent's positions. Layer four marks the position of bombs with ones. The explosion map of the game state is present in layer five. This uses the same representation as the one passed from the game. The sixth layer encodes the positions of coins with ones.

1.4 Models (Patrick)

Throughout the development phase, we mainly used two types of models. In the following we describe their structure, their functionality and why we decided to implement a more complex model.

1.4.1 DQN

As mentioned in section 1.3, our initial idea was that convolutional neural networks would be suitable models for the task. For this reason we implemented a Deep Q-Network (DQN) similar to the approach presented in [Mn15] with slight modifications to adapt it to our task. The model consists of a convolutional layer of size 32 with kernel size 2 and stride 1. This layer is followed by another convolutional layer of size 64 and a kernel size of 2 and stride 2. The third layer is also convolutional and its size is again 64, reducing the stride to 1. The two following layers are linear layers of size 512 and 6, which is the output of the network and presents the action predicted as a vector of length 6. Between all layers we are using also ReLU layers.

1.4.2 ADRQN

We encountered several limitations of the initial DQN model that slowed down the training process: For example, when navigating the game field, the agent showed behavior of repeating the same steps in endless loops. Our explanation for this phenomenon is that the lack of awareness of past steps creates a problem, if the agent has not learned which action yields a significantly better return. The DQN agent is only aware of the current game state at every step, which hinders learning good behaviour in those situations. For this reason, we decided to look for a model which is aware of past steps and by taking not only the current game state, but also past steps into account, relaxes the Markov property.

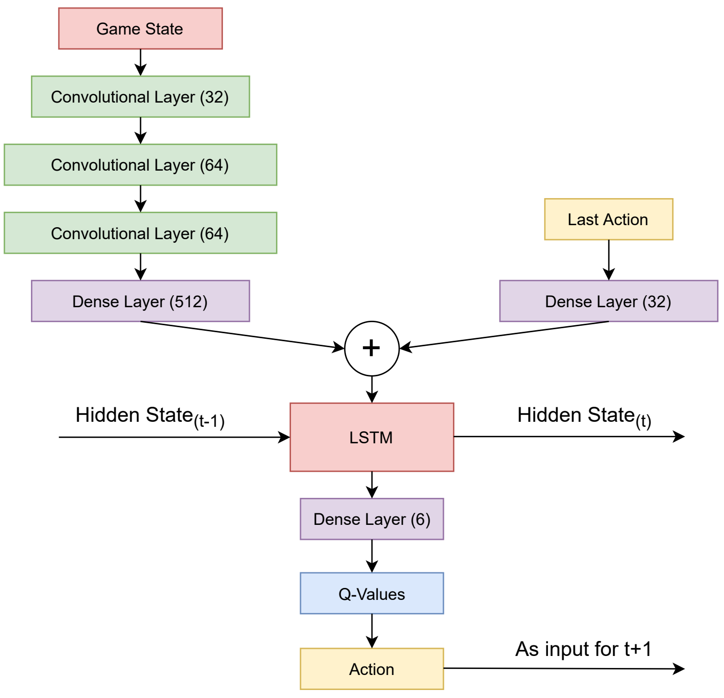

Figure 1: Architecture of our ADRQN model

This second model we implemented is called Action-specific Deep Recurrent Q-Network (ADRQN) and based on the approach presented in [Zh18] and an example implementation from [Pe20]. However, we had to change our model significantly from the architectures of these approaches, to make it feasible for our task. The overall structure of the ADRQN is a neural network which has a series of game states as its input and the corresponding previous actions as one hot encoded vectors which are fed through a linear layer. The outputs of both are then concatenated and used as the input to a Long short-term memory (LSTM), along with the hidden state of the LSTM at the previous step. The LSTM's output is then fed into more neural network layers which output the predicted action. Instead of predicting actions to sampled single game states like the DQN, training the ADRQN is done by feeding batches of sequences of subsequent game states into it.

We experimented with different layer types and sizes to implement the ADRQN, since the presented approaches turned out as unable to learn in our task. We therefore tried to use the architecture of our DQN network as input for the LSTM and embedded the previous action through one dense linear layer of size 32. The output of the LSTM is fed through one dense linear layer to output the predicted action as a vector of size 6. This early approach we tried did not perform well and could not increase rewards through training in multiple episodes. We discovered that the problem was the input size to the LSTM being too small, since we only increased the fifth layer to an output size of 64. For our final model, which is visualized in Figure 1, we omit the fifth layer completely and use the output of layer four with a size of 512 directly as input to the LSTM. We consider this model suitable in terms of complexity and flexibility for our task. Furthermore, we expect it to be fast enough to make decisions below the time threshold. We measured an average execution time of the *act* method of 0.003 seconds on our local machine, which is equipped with an Intel Core i9-9880H.

1.5 Tuning Hyperparameters (Patrick)

Our models require several hyperparameters to be determined to ensure being able to learn. In this section we briefly describe the most important parameters we chose and their influence on training.

An important parameter for Q-Learning is the discount factor γ. We found that in most literature this value is set near to 1, to take future rewards into account strongly when optimizing the model. In our own testing, we discovered that for our task and reward structure setting γ to 0.999 yielded the best results. Our initial intuition for tasks where we implemented rewards, so that actions always resulted in immediate rewards, was that a smaller γ would lead to faster learning. This could not be confirmed experimentally. A γ near 1 lead always to better results in terms of rewards per episode.

Another hyperparameter that turned out as very important is the number of episodes after which the target model is updated with the weights of the policy model. Here, we found that this parameter leads to a trade off between learning speed and stability. If the target model is updated too often we risk that the policy model did not improve or decreased prediction quality. In this situation, the target model makes lower quality predictions when updated, leading to a decrease in policy prediction quality over subsequent steps. On the other hand, choosing a too big number of episodes between updates makes the increase of prediction quality of the target model slow, which decreases learning speed, or as we also could confirm experimentally, destabilize training. Therefore, our goal was to choose a target update that was as small as possible, but does not destabilize the learning process. We found that for the DQN model a target update for every ten episodes was sufficient. For the ADRQN we had to choose a target update of at most every 20 episodes, which we had to increase to 30 episodes when learning more sophisticated tasks.

In terms of batch size we found that in most literature big batches are considered to be often helpful for training in Q-Learning [SA19]. For us the limiting factor for possible batch sizes was the limited amount of hardware we were able to use. Especially for the ADRQN training, large batch sizes increase the time per game step significantly, since each data point in the batch consists of a sequence of sampled steps. We found experimentally that a batch size of 64 is reasonable fast.

[Zh18] states that sampling large connected sequences of game states for each data point for training the ADRQN is beneficial. The length of this sequences again affects the time for each optimization significantly. We decided that a length of twelve transitions yields the best trade off between training speed and quality of policy predictions for us.

2 Training Process (David)

This chapter should explain how we did the training. It shows gradually the progress in training and how we tried to improve our model to accomplish the final task, playing a real bomberman match.

2.1 Reward Engineering (David)

In this section the self-invented rewards are described. It shows how they are computed and which problems they solve or which learning approach they handle. The different rewards will be mentioned later again, when the training process is described. We invented them time by time, depending on the requirements we faced.

2.1.1 Coin distance

Since coins are the only way to get points in the game, besides killing opponents (which is the much harder task for the agent), it is very important to train the agent collecting every coin, which is visible and reachable. This should be highly prioritized, much more important than bombing crates away, because in first place this gains no high score.

To achieve that the agent is moving towards coins, a reward is needed, which steers the agent in coin proximity. Although there is a pre-implemented event for collected coins, another reward helps the agent to get near the coins. Collecting a coin is a very rare event for the agent, because at the beginning of the round there are only nine coins on the field. So its very hard to find those. If also the distance to the nearest coin is computed and rewarded, the agent has a lot more reference points to find the next coin on the field.

Therefore we computed the euclidean distance between the agent and the nearest coin which is visible. The distance is normalized between zero and one and so the following function calculates the associated reward:

$$R = r_{max} - d_{Coin} \cdot r_{max} \tag{2.1}$$

Notice that the reward is changeable by the variable r_{max}, so it can be varied depending on the importance of collecting coins in the respective training. Since d_{Coin} is between zero and one, you get a reward of r_{max} if the distance is zero and a reward of zero if the distance is one (the agent is on the very other side of the field).

2.1.2 New position discovered

To find all the coins on the field, it is important that the agent is discovering the whole pitch. Moreover it is common that the agent keeps moving in the area of the field, where he is spawning.

To solve this problem we added a new event *NEW POSITION EXPLORED*. It should be appended to the event list, if the agent is in a position, where he did not move before. Therefore a empty list is initialized in the setup. If the position on which the agent moved is not in the list yet, a x-y-tuple of the current position is appended to the list. Bit by bit a list with the already explored tiles is created. Now a reward for the event can be easily assigned and the agent should be forced to explore the whole field.

2.1.3 Repeated steps

A major problem we faced during training is that the agent is often repeating his steps. Especially at the end of a round, where less random actions are used, he gets stuck in one position or one sequence of moves. If resting in one position gains a lot of reward (e.g. near a coin) in relation to do some other stuff, the agent starts repeating his steps or is waiting for the rest of the time.

The idea is that the last three positions are considered to see, if the agent is changing his direction. The goal is that he can take maximum of three steps in one position. After that he gets a negative reward, which should lead him to move to another tile. Therefore a list with the positions is initialized, like we did with the discovery of new positions. The difference is, that the first entry of the list is popped, if its length is higher than three. This means that only the last three positions are considered. If the agents position equals one of the last three positions, he had repeated his steps and a negative reward is given. A possible approach could be that more than three steps are considered, if the agent has a bigger movement space in which he repeats his steps.

2.1.4 Destroyed crates

At the beginning of the game the agent mostly has only three positions to move on. The rest of the field is covered by crates or stone walls. To reach any coins or opponents to blow them up, it is important that the agent is creating more space for movement by placing bombs near crates. Although there is already an event *CRATE DESTROYED*, we decided to invent a new reward for two reasons.

One problem is that the event *CRATE DESTROYED* only appears, when the bomb explodes and the crates are destroyed. Since the drop of bomb happened four time steps before it is very difficult for the agent to realize these two events belong together. To give the agent a better understanding of combining these two events, the new reward for destroyed crates is already calculated, when the bomb is dropped. We calculate which crates, the bomb will destroy in the future instead of waiting four steps and giving the reward then. The second problem with the pre-implemented event is that only says *that* one or more crates are destroyed and not *how many*. Because a bomb is better placed, if destroying more crates, we will also consider how many crates are destroyed with one bomb. All of this leads to the conclusion that a new self-invented reward for destroyed crates is needed to get good training results.

To compute the number of crates, which will be destroyed by a bomb, we can use the the field variable in the game state, where every crate is marked with a 1. We have to loop through all the positions, which are one to three tiles away from the bomb in every direction (bomb explosion radius). If the x-y-coordinate equals one in the field variable, we can add one crate to the total number of crates. It is very important that the calculation is stopped, if the x-y-coordinates value is minus one, which means there is a stone wall that blocks the explosion. The number of destroyed crates is only calculated, when a bomb is dropped, so destroying crates is only rewarded once. Depending on the specific training the reward of one destroyed crated can be varied.

2.1.5 Bomb range

One of the most import things during the match is, that the agent is aware of getting in bomb range, because this often means the end of the round. To train the model to not get in bomb range, we computed a reward, which is negative, if the agent gets in the explosion radius of a placed bomb. To simplify the computation it is not necessary *when* the bomb is exploding.

To compute this reward the x- and y-coordinates of the agent and the bomb are compared. Every position from three tiles up, down, left and right of the bomb are compared with the agents position. If the agent is in bomb range, the method returns true. A very important aspect which we found after a small number of rounds is, that we did not pay attention on stone walls between the bomb and the agent, which obviously block the bomb explosion. So we implemented another restriction, which checks if a stone wall lies in the explosion range.

2.1.6 Bomb distance

We have a reward which tells us if the agent is in bomb range and therefore in danger, but to direct the agent out of this range another reward is needed. The distance between the agent and the bomb is a good indicator for this, because the further away he goes, the less likely it is, that he bombs himself or dies by the bomb of a victim. The idea is that the agent gets a negative reward if he is very close to a bomb and the reward is falling with more distance to the bomb.

The computation is very similar to the computation of the coin distance. So we compare the agents position with the position of a placed bomb. It is important that the reward is only given if the agent is in bomb range. This prevents a highly negative reward in case that the agent is moving around a corner. He is still very close to a bomb, but should not be punished for this movement, because he steered himself out of bomb range.

Since the distance between the agent and the bomb is normalized, the reward has to be calculated with the following equation:

$$R = \frac{d_{Bomb} \cdot 21,5 \cdot 1000 - r_{min}}{r_{min}/1000} \tag{2.2}$$

$d_{Bomb} \cdot 21,5$ gives us the total distance between the agent and the bomb (back calculation from the normalization). r_{min} should be in thousands range. For example if the r_{min} equals 4000 the total reward for a distance of two is -500 and for a distance of three it is -250. The reward should be set higher, if the agent can not do it out of bomb range.

2.1.7 Bomb in corner

Most of the time a bomb in a corner is suicidal for the agent, because he has no chance to dodge the explosion. Surely he can do this at the end of the game, but in the beginning his path is blocked by the crates. Therefore we defined the coordinates of the four corners of the field. If the the position of the agent equals the coordinates of one of the corners and the action is "bomb dropped", a highly negative reward is given. This results in the agent not dropping any bombs in any corners.

2.1.8 Reward normalization

We also started to normalize the reward in the very beginning of the project. For us as developers it is easier to give natural numbers as rewards, because we could imagine better how we should adjust the rewards. For better convergence of the model a normalized reward is more advantageous, because the activation function work faster with rewards between zero and one. Therefore we had to normalize the reward, so that we can give rewards in the thousands range and have a better sense for the weight of the rewards. In the end of the reward method we transpose the total reward between zero and one, so the neural network can work with it well. Since the maximum reward is defined strictly between 0 and 1, the normalized reward can also be higher than one in some cases. This does not affect training significantly, since it only deviates slightly.

2.2 Training procedure (David)

In this section the training procedure is described. We split up the training in different milestones like it was recommended final project task. This section should also show which problems we faced during training and how we solved it by e.g. adding new self-invented rewards.

2.2.1 Collect coins

First of all we tried to teach the agent how to collect coins on an empty field. Means that there are no crates and opponents, only coins that need to be collected. The goal of this milestone is that the agent has a purposeful movement around the field and always is trying to move towards coins. If the agent is able to move around the field and collects around 90 percent of the coins in 400 steps, we can move on with the next milestone.

The first approach is to reward only the event of *COIN COLLECTED*, so the only way to gain a high reward is to collect the coins on the field. We train the model for 400 rounds and see that he is not learning anything. The random action to drop a bomb is removed, so the agent only focuses on the movement and collection of coins and is not distracted by random bombs, which will kill him. Most of the time he only stays in one position.

Because there are only a few coins on the field, the event *COIN COLLECTED* is very rare. So we raise the number of coins on the field, that the agent has more sense of achievement. Additionally we invent a reward for the distance to the nearest coin, as described before. We try to generate more situations where the agent is collecting a coin, so he can learn from it. The agent has a higher to incentive to navigate into coin direction now. After hundreds of training rounds we see first successes. The agent is strictly moving in coin direction, although he sometimes stays in one position, especially when he is exactly between to coins. With some random actions, which give new impulses he continues the coin hunt.

Since the agent is now able to collect coins and moves around field effectively, we can move on with placing crates on the field and teach the agent how to react. Because the following processes are much more complex and the agent needs more rounds to learn, we switch to Colab, to get a faster training. As we describe in the third chapter, we weigh up between a DQN and ADRQN model now. The following training continues only with ADRQN.

2.2.2 Bomb crates

Since the agent is now able to collect the coins effectively, we now try to teach him how to act on a field with crates. We want the agent to destroy crates targeted. If a new coin is exposed, he should fall back on what he learned how to collect coins effectively. The biggest challenge is to place bombs, but do not kill himself.

First we start the training by giving a reward for placing a bomb, since the agent has not been able to set bombs since now. We hope the agent places bombs and then moves away, because we give a highly negative reward for killing itself. We do not give any rewards for other movement, trying the agent only focusing on setting bombs and get away. The agent should be very free in his decisions in the beginning. After 400 rounds of training the agent is still not setting bombs, probably because the reward for setting bombs is too low. Another problem we face is that the agent can not dodge the randomly set bombs constantly. Therefore we raised the reward for setting bomb, either reduce the reward for killing itself.

After another 400 rounds of training we see that the agent has not really learned something new. Only giving bomb reward and negative reward for killing itself is probably not enough information for the agent to improve. Like described in the last section we now add some self-invented rewards to give the agent more reference points for improvement. We come up with a reward for destroyed crates, which should make it more likely that the agent is placing bombs, especially near crates, since he now has a higher incentive by gaining higher reward. The other reward we add is the in-bomb-range-reward. Inasmuch as the agent is not able to cover himself from explosions, we try to steer him out of bomb range with this negative reward. Another 1000 rounds of training show that the new rewards really help the agent to act more in a logical way. Although he still not places bombs himself, he is moving onto positions where a bomb plant could be profitable. He also managed to get out of bomb range in a few rounds. In retrospect it may be better to add the new rewards step by step, because we can not really see which reward has which effect on the behaviour of the agent. So after these rewards we only add or change one reward, clearly seeing what changes it makes to the model.

To train the agent to plant bombs himself we raise the rewards for bomb planting and destroyed crates again. Improving the ability to move himself out of bomb range, we give a new reward to the model. In addition to the in-bomb-range-reward, we add the reward for the bomb distance as described in the first section of this chapter. One problem is that the negative reward of the bomb distance and the bomb range will lead to an agent who is not dropping any bombs, because it does not pays off for him. So the reward for dropping bombs has to be very high. After 1000 rounds we see first results and the agent manages to move out of bomb range very constantly, although he is still not able to plant bombs himself. We also observe that he is not collecting a coin if it is revealed.

What we also see is that the agent is not gaining more rewards in the end of the training. This suggests overfitting respectively not enough training data of what he can learn. If all of the rounds during the training look pretty the same (e.g. he is waiting for 400 steps), the agent has no opportunity to gain new knowledge. So we decide to use more random actions. Until now we always had a relatively low randomness in our training. By raising the random factor, we hope to show the agent more possible movements and help him to find better ways to play the game. We do thousands of training rounds and can observe that the training reward is improving very well. Also the replays show that the agent tends to do more movement and is not waiting in one position for a long time.

Hence the agent is now dodging bombs very well, but often stays in one corner of the field, we bring back the step-repeated-reward, which we have introduced during the coin training. We want the agent to move all over the field and finding new areas on the pitch, so he knows what do after a few steps, when there are less crates. Unfortunately we see that the agent always starts to repeat his steps no matter how high the negative reward is. Therefore we decided that a low number of random actions is important, even in the real match, so he gets a new impulse if he is stucked in a position. Even though we often raised the reward for bomb dropping, this action is really rare, so we need some random bombs to be placed. If a bomb is placed randomly he is able to use this action effectively.

Since we have a very confident agent right now we want to start training him for a long time. One idea to get better training data is to filter invalid actions during the training. We implement a method that checks, if a chosen or random action is valid. If this is not the case another has to be chosen. We hope that the game examples are more meaningful for agent with this method. Although invalid actions are punished by rewards he often decides to do them or more annoying, they are randomly chosen. Particularly when a bomb is dropped a invalid action means suicide, because he needs all of the next for steps in the right direction to get out of bomb range. We see the total rewards rising in the next 2000 rounds of training, what shows us that the filtered actions help the agent to get better training data.

The agent is now able to drop bombs, although we also need random bombs, because he is a bit shy to plant them himself. If a bomb is dropped, he dodges the explosion radius very well. To strengthen the agents behaviour we now train him for thousands of rounds, always supervising his moves and rewards. A little issue is that he does not collect coins anymore, so we raise the coin- and coin-distance-reward step by step during these final rounds of training. The agent is now ready to compete with opponents and learn how to fight them and not be killed by them.

2.2.3 Play against agents

The last part of the training is to play against other agents. Here the agent should learn how to place bombs to trap opponents and how to dodge not only the own, but also the opponents bombs. It is important, that the agent can move to disclosed coins very fast, because he has to reach them before an other agent does. The last trainings all had an empty layer in the feature vector, where the positions of the other agents are pictured. Therefore the model has to stable to not get unsure about decisions, because of these new information it gets.

Due to time issues, we can not train the agent to compete against other agents for a long time. Nevertheless he fulfills the purpose of the bomberman game, as he learned to place bombs correctly, move out of bomb range and collecting disclosed coins. The challenge to kill other agents is probably the hardest one. Presumably it will need a lot of new computed rewards and a adaption of the architecture to teach the agent how to move in opponent direction and trap them with a placed bomb. Of course this ability gives the highest score by far.

3 Experiments (David & Patrick)

Having described the training process in chapter 2, we discuss several experiments we did and modifications to our training structure we tried out. We explored several different modifications to our approach, but focus on the most important aspects for the overall approach in this chapter.

3.1 Experimental and Training Setup (Patrick)

Our approaches rely on neural networks, with the ADRQN being a rather complex model, training on CPU was not feasible. Since we do not have access to GPUs our own, we had to use Google Colab[1] for training. Executing the game code on colab was rather intricate, because of incompatibilities, but speed up the training process significantly. Furthermore, we experienced significant limitations due to usage limits. For this reason our main strategy during the development of rewards and tuning of hyperparameters was to monitor metrics like the progress of rewards per episode and steps done per episode. Because of the time limitations we discarded training early, if we experienced tendencies of decreasing rewards over several hundred episodes or too strong stagnation of rewards.

3.2 DQN vs. ADRQN (Patrick)

As mentioned in section 1.4.2, we started the development phase with a DQN model, but then also experimented with the more sophisticated ADRQN approach. The main reasons for this are two major limitations of the DQN we encountered. In the following we will discuss these, and compare both models on their results of the first training for the coin collection task.

3.2.1 Limitations of the DQN model

Even though the DQN model was able to learn collecting more coins over the course of episodes, by examining the performance of the model playing multiple episodes,

[1]https://colab.research.google.com/

we found out that the agent would get stuck moving between two fields. The reason for this is that we used the coin distance approach described in section 2.1.1. When we inspected the reward gained at each step, it got clear, that the neighboring fields that caused this phenomenon caused similar and relatively high reward, since they had almost exactly the same distance to the nearest coins. The cause for this problem got more clear when we tried to overcome it by introducing negative rewards for the agent visiting one of the last three fields it came from. Our implementation of the DQN had no memory, being based around the idea of the game having a strict Markov property, thus being not able to recall last steps. Furthermore, we expected this to be hindering in the context of placing bombs. We wanted to ensure that a learning of the connection between placing a bomb and the following explosion in the future would be possible.

3.2.2 Comparing the Approaches on the Coin Collection Task

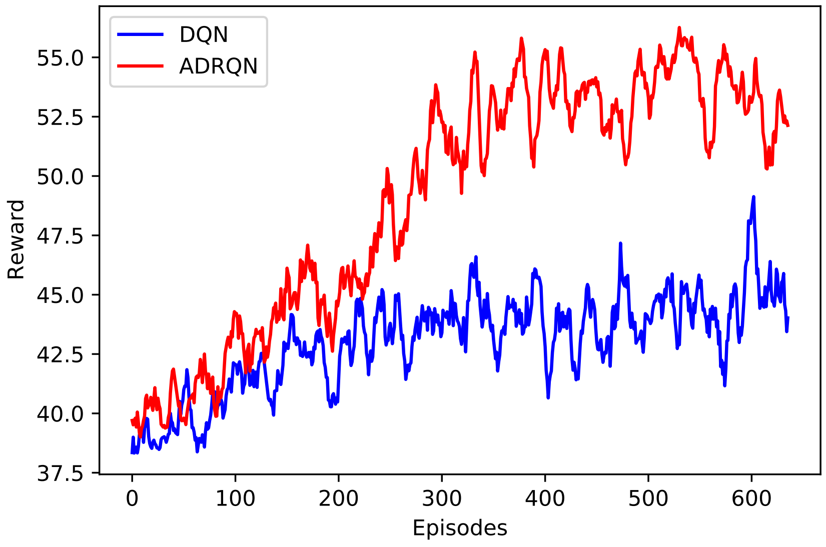

Figure 2: Rewards per episode on the first training episodes of the coin collecting task

Because of time limitations and limited hardware resources, we compared the DQN and ADRQN model only in early stages of the development process. We decided to use the ADRQN for further training processes, because of its faster learning in terms of reward gained per episode. Figure 2 visualizes the reward per episode during initial training over 660 episodes, using a moving average of ten episodes for better readability. In this scenario, only coins, without crates or adversarial agents were present. It is clearly visible that the agent using the ADRQN model outperforms the one with the DQN. After around 450 episodes, the ADRQN agent already achieves over ten reward points more than the DQN. We were able to confirm the superiority of the ADRQN agent when running both agent at the same time for testing. In this environment the ADRQN was able to collect significantly more coins than the DQN.

3.3 Random threshold (David)

During the training for destroying crates, we worked with different methods to determine the randomness of actions. Since the randomness of actions during the training is very important to give the model new inputs, we decided to weigh up between a constant random threshold and an epsilon greedy threshold. The threshold indicates which percentage of the actions are chosen randomly during the training.

An epsilon greedy threshold describes a variable threshold. With a start and an end point the threshold falls steadily with an increasing round number. So in the beginning of the round the actions of the agent are very random, but in the end he is on his own. This gives new impulses to agent, but he can also have time to do his own movement. In our case the threshold starts with 40% and goes down o 10% in 50 steps. After that it is set back to 40%. The constant random threshold never changes during the round, as the name says. I should give the agent new impulses all the time. Since the epsilon greed threshold is about 20% on average, we decided to set the constant random threshold to 20% as well.

We do the comparison after 8000 rounds of training with crates. As you can see in figure 3 the average reward per round is always higher with the epsilon greedy threshold. An explanation for this can be, that the agent needs the rest periods, where he is only doing his own movement, to try things out without getting disturbed by a random factor. Also the epsilon greedy threshold starts with a noticeably higher randomness of actions. The agent gets a lot of new impressions in this period. For the rest of the training we continue with the epsilon greedy threshold.

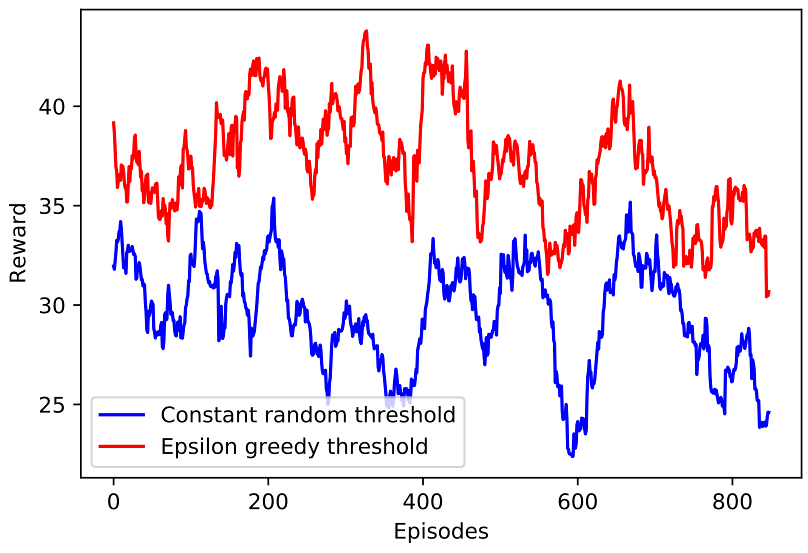

Figure 3: Rewards per episode with different random thresholds after 8000 rounds of training

3.4 Variations of the Replay Memory (Patrick)

As described in Section 1.4.2, the ADRQN model is trained by sampling sequences of, in our case, length twelve from the replay memory. During the training with crates, we initially experienced the agent to kill itself often and at the beginning of episodes, thus feeding very short sequences of transitions into the replay memory. Our intuition was that this could negatively affect the learning process, since it is likely that sequences including multiply episodes are sampled from the replay memory. Our initial implementation of the memory only incorporated a flag for final transitions of episodes to then not predict future Q values after those transitions. To further encourage learning from sequences of same episodes and thus being able to predict Q values that lie in the far future, we experimented with a modified version of replay memory, that only samples sequences of the same episode. We tested this new version along with the first implementation. While still being able to create a working training process, the episode specific replay memory did not improve the reward per episode, nor the number of steps the agent was able to perform. We discarded this change, because it introduced the problem, that many episodes had to be simulated before being able to form large enough batches holding only sequences of transitions from the same episode. Since we could only train in limited time frames due to Colab's usage limits, this meant that for every training a large amount of non improving sequence samples were present in the replay memory at the beginning of the training.

4 Conclusion (Patrick)

4.1 Possible Future Improvements (Patrick)

The main limiting factor for our development was the limitations on available GPU hardware. Because we relied on Colab for training, we had to deal with the usage limitations that come with it. Thus, we had to do training in sessions of around two hours before having to start a new training process. This not only meant that training was labor intensive, since we had to restart it multiple times a day, but also meant that the replay memory started empty for every training session, giving the model only limited diversity in samples when starting the training process. We believe that our model and training infrastructure are working and can lead to very good results, but also think that much more training time should be invested to obtain its full potential.

Furthermore, we had to invest quiet a long time in hyperparameter tuning, but still think that further optimizations could be made in this context. If we had no time or training hardware limitations, we would further compare different hyperparameter settings and model architecture variations on rather basic tasks like coin collecting to ensure optimal conditions for more sophisticated tasks later on. We would also try automated changing of parameters over the course of episodes, like changing target update cycles, when the agent learned enough to perhaps have robustness to updating more often and therefore possibly speeding up the training.

4.2 Feedback (Patrick)

We found the game to be very feature rich with a well defined structure, which made developing an own agent fun and very encouraging. Because of the last few lectures before the project, we felt well prepared to also explore more sophisticated reinforcement learning methods.

The only major difficulty we faced during the development was in terms of compatibility regarding the requirements of the game. From the beginning, we planed to use Google Colab to train our models, since we do not own hardware that could be able to train our convolutional neural networks with reasonable speed. The problem we encountered was that Colab only supports python 3.7 per default. Installing the version 3.8 which the game does require was not an easy task and required some rather unusual workarounds. Furthermore, using python 3.8 on Colab also requires installing all python packages every time we started the notebook, and also causes incompatibility with many offered GPU types. Since Colab and Kaggle Notebooks will update their python version in the future, this problem probably will not exist anymore next year, but this incompatibilities should possibly be considered in future versions of the game.

4.3 Code Repository

Our code repository can be found at https://github.com/theGindar/bomberman.

Bibliography

[EM04] Even-Dar, E./ Mansour, Y.: Learning Rates for Q-Learning. J. Mach. Learn. Res. 5/, pp. 1–25, 12/2004.

[Fa20] Fan, J./ Wang, Z./ Xie, Y./ Yang, Z.: A Theoretical Analysis of Deep Q-Learning. In: Proceedings of the 2nd Conference on Learning for Dynamics and Control. Vol. 120. Proceedings of Machine Learning Research, PMLR, The Cloud, pp. 486–489, 10–11 Jun/2020, URL: http://proceedings.mlr.press/v120/yang20a.html.

[Mn15] Mnih, V./ Kavukcuoglu, K./ Silver, D./ Rusu, A. A./ Veness, J./ Bellemare, M. G./ Graves, A./ Riedmiller, M./ Fidjeland, A. K./ Ostrovski, G./ Petersen, S./ Beattie, C./ Sadik, A./ Antonoglou, I./ King, H./ Kumaran, D./ Wierstra, D./ Legg, S./ Hassabis, D.: Human-level control through deep reinforcement learning. Nature 518/7540, pp. 529–533, 02/2015, URL: http://dx.doi.org/10.1038/nature14236.

[Pe20] Peschl, M.: Reinforcement Learning (DQN) Tutorial, 2020, URL: https://mlpeschl.com/post/tiny_adrqn/.

[Py17] PyTorch: Reinforcement Learning (DQN) Tutorial, 2017, URL: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html.

[SA19] Stooke, A./ Abbeel, P.: Accelerated Methods for Deep Reinforcement Learning, 2019, arXiv: 1803.02811 [cs.LG].

[WD92] Watkins, C. J. C. H./ Dayan, P.: Q-learning. Machine Learning 8/3, pp. 279–292, 05/1992, DOI: 10.1007/BF00992698, URL: https://doi.org/10.1007/BF00992698.

[Zh18] Zhu, P./ Li, X./ Poupart, P./ Miao, G.: On Improving Deep Reinforcement Learning for POMDPs, 2018, arXiv: 1704.07978 [cs.LG].