Report

Chernenko, Tatjana Mylius, Jan-Gabriel Wüst, Valentin

25.03.2019

1 Introduction

This report aims to document our approach to training an agent for the multiplayer game Bomberman using reinforcement learning. Roughly following the approach taken in [1], we decided on using deep Q-learning [2] with a dueling network architecture [3] and separate decision and target networks. To improve the training process, we utilized prioritized experience replay [4]. Finally, we further enhanced our model to explicitly exploit the inherent symmetries of the game, the translational symmetry that is incorporated by using convolutional layers in our network, and the rotational and mirror symmetries that are enforced by symmetrizing it [5].

The code for our agent can be found in our GitHub repository:
`https://github.com/valentinwust/IFML-Bomberman`
Our agent utilizes, among others, the libraries keras and tensorflow.

2 Model (Jan-Gabriel)

In the following chapter we will first give an overview of the game, then we will explain appropriate techniques for creating a self-learning agent, outline our reasons for choosing or rejceting them and finally describe the resulting architecture.

2.1 Bomberman

Bomberman is a maze-based strategy game. First released as Bakudan Otoko in Japan in 1983. This project's objective is to create a self-learning AI for the multiplayer variant. In this variant the goal is to collect as much coins as possible, kill the enemies and stay alive. The four players are placed in the corners of the maze. Each player can perform six possible actions: left, right, up, down, place bomb, wait. The board consists of indestructible walls, and crates which can be destroyed by bomb explosions. Additionally, coins are scattered across the board and can be collected to gain points. At the beginning of the game, the players are separated by crates and most coins are hidden underneath

crates, thereby requiring the players to destroy the crates to win. Each round ends after 400 timesteps.

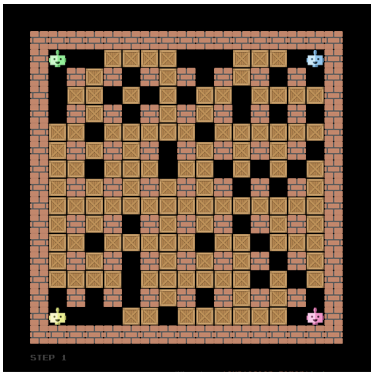

Figure 1: The board with 4 players in the starting position.

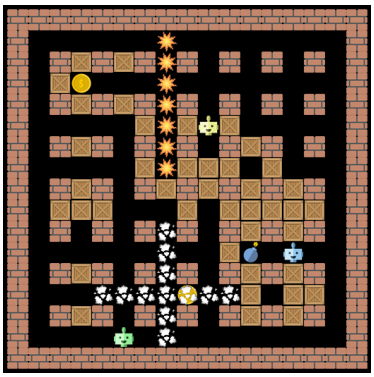

Figure 2: The board with 3 players playing.

As can be seen in Figure 1, the playing field consists of a 17×17 grid. Of those 289 tiles, 113 are walls and therefore static. This effectively leaves 176 tiles on which the games is played. A total of 9 coins and 132 crates is randomly placed on them, for the crate placement three tiles in every corner are excluded, while the coins are hidden behind the crates. Additionally, there are at most four players, bombs, and centers of explosions. The players can be in two different states, i.e. being able to use a bomb or not, and bombs and explosions have four and two different states, respectively. The amount of possible starting states with four players gives an impression of the size of our total state space:

$$\binom{164}{132} \times \binom{132}{9} \times 4! \approx 6.80 \times 10^{48}$$

2.2 Reinforcement Learning

Reinforcement Learning is a technique going back to the 1950s, probably originating with Richard Bellman and his paper "The theory of dynamic programming" [6]. The problem to solve was, how to design controllers that minimizes a certain measure of a dynamic system, so called optimal controller problems. We can model problems of this nature by asserting that we have states s, observations about these states o, take actions a and get rewards r. As can be seen in Figure 3 we make an observation o_t of the current state s_t, take an action a_t, resulting in a new state s_{t+1} and get a reward r_{t+1} based on the outcome. States are not necessarily only affected by our own actions, but can be affected by other factors, in our case the enemies' behavior.

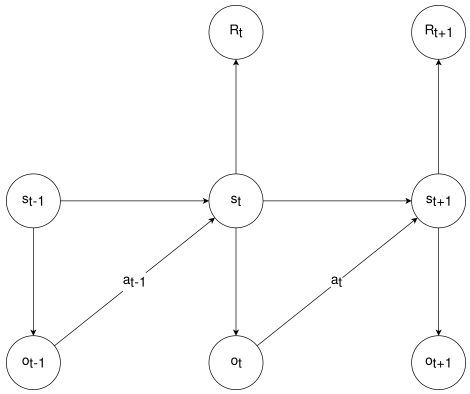

Figure 3: A model of the state transitions.

2.2.1 Q-Learning

Our first instinct was to implement Q-learning. In Q-learning decisions are made by looking up the expected reward of an action in the Q-table. The Q-table assigns every action at every state a value – the expected reward. After every action the agent uses the received rewards to update the Q-table for future reference. The values are updated according to the Bellman equation [6]:

$$Q(s,a) = (1-\eta)Q(s,a) + \underbrace{\eta}_{\substack{\text{learning}\\\text{rate}}} \Big[\underbrace{r}_{\text{reward}} + \underbrace{\gamma}_{\substack{\text{discount}\\\text{factor}}} \underbrace{\max_{a'} Q(s',a')}_{\substack{\text{estimated optimal}\\\text{future reward}}} \Big] \quad (1)$$

Here, s' is the next state after taking the action a in state s. The discount factor ensures that rewards received earlier are weighted more than those received at a later point, and the learning rate determines how much each fit changes the existing Q-values.

Using the expected reward of the next best action to fit our values ensures that the Q-values represent the discounted reward for following the current

policy until a terminal state is encountered, since the reward for the next action again contains the discounted reward for the action after that and so forth. If there is no terminal state, the Q-values still converge due to γ being smaller than one.

$$Q(s_t, a_t) = \sum_{i=0}^{T} \gamma^i R_{t+i}(s_{t+i}, a_{t+i}) \tag{2}$$

Due to the size of the necessary Q-table we decided against using simply Q-learning, we instead used deep Q-learning. Here, a neural network is trained to estimate the Q-values for all states. This has the obvious advantage that the neural network (hopefully) generalizes to unseen states, while a normal Q-table does not.

2.3 Deep Reinforcement Learning

We now have to address two problems, the rather large state size and that there are some actions for which only the relative position on the board is relevant. In other words some actions are translationally invariant. Convolutional Neural Networks allow us to solve both of these.

2.3.1 Neural Networks

Artificial Neural Networks are frameworks inspired by biological neural networks. As can be seen in Figure 4 they consist of at least three layers:

Input Layer Loads the data to be processed into the system

Hidden Layer Between the input and output layer, weighs different inputs to produce an output

Output Layer This layer creates the outputs.

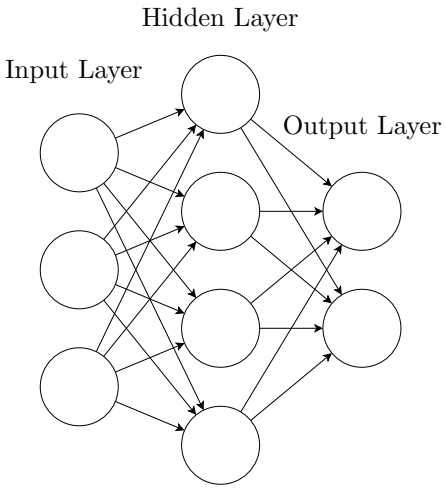

Figure 4: A neural network consisting of a 3-node input layer, a 4-node hidden layer and a 2-node output layer.

Single layer neural networks, also known as perceptrons, are only able to solve linearly separable classification tasks. More complex neural networks containing several hidden layers are able to solve regression problems and complex classification problems. So how does a neural network actually work? To better understand this, let us take a look at the function of a single neuron, which is given by:

$$\underbrace{o_k}_{\text{output}} = \sum_j \underbrace{i_j}_{\text{input}} \cdot \underbrace{w_{jk}}_{\text{weights}} + \underbrace{b_k}_{\text{bias}} \tag{3}$$

The output is calculated by summing over the weighted inputs and adding a bias. During training the network is fitted to a desired output by adjusting the weights given to the individual inputs. However, biological neurons don not just give a weighted output, but rather sometimes decide to fire and sometimes don't, this creates a non-linearity. To imitate this behavior, an activation function is added to the neurons. We decided to utilize a rectified linear unit (ReLU) function (See: Figure 5). This was done because the ReLU function offers the advantage of sparsely activating, which means fewer neurons are active. This translates into a noticeable performance boost.

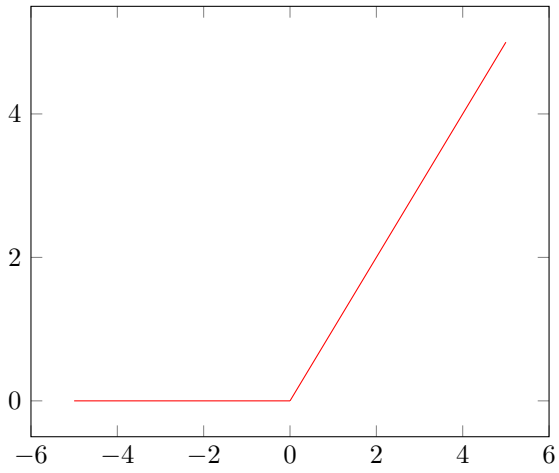

Figure 5: Recified Linear Unit Function

However problem known as the "dying ReLU" can occur. When a neuron learns a strong negative weight, it will not activate again. There are ways to counter this problem such as leakyReLU functions or exponential LU functions. Testing these showed no difference, leading us to assume that we didn't run into this problem.

Convolutional Neural Networks A convolutional layer uses a stack of filters that stride over the input vector. Each filter returns a single output value for every slice of the input. In our case, the input consists of an array with L channels, and $M \times N$ spatial directions. For K kernels of size $L \times P \times Q$, and strides S_P and S_Q, the output then has the shape $(K, \frac{M-P}{S_P}+1, \frac{N-Q}{S_Q}+1)$. Of course, the kernel size and strides have to be picked in a way that fits the shape of the input. The output is then given by:

$$\underbrace{o_{k,u,v}}_{\text{output}} = \sum_{l=0}^{L-1}\sum_{p=0}^{P-1}\sum_{q=0}^{Q-1} \underbrace{\omega_{k,l,p,q}}_{weights} \cdot \underbrace{i_{l,p+u\cdot S_P,q+v\cdot S_Q}}_{input} \tag{4}$$
$$u \in [0, \frac{M-P}{S_P}], \quad v \in [0, \frac{N-Q}{S_Q}]$$

Since every filter gets shifted over the whole input, the translational symmetry of our input is maintained in the output.

2.3.2 Symmetrization

We already use convolutional layers to exploit the translational symmetry of the state space, but the state space also has a rotational symmetry. If the actions are adjusted accordingly, a rotated version of the state should lead to the same

action. Additionally, the mirror image of the state should also return the same action. Of course, the resulting action needs to be adjusted according to the operation performed on the state.

We first tried to add every experienced transition to the memory multiple times in all rotated versions. However, then our model still needed to learn all versions of the state separately. To circumvent the need for this, we forced it to predict the same action for every equivalent state.

This was done in a similar way as in [5]. First, we calculate all eight rotated and mirrored versions for every state our model receives. For this calculation, additional layers were inserted into our model. To limit the necessary computations, this is done sequentially. Every 17×17 matrix in our state is multiplied by the rotated unity matrix, producing the mirror image along a vertical line through the center of the arena. The resulting matrix is then transposed, producing a version of the original state that is rotated by 90 degrees to the left. Here, we used that a rotation can be generated by mirroring twice. This is repeated until all eight versions are generated.

Afterwards, all equivalent versions are fed through our model as described below. The resulting Q values are then permuted and averaged, the permutation being necessary to reverse the mirroring and rotating of our state.

Doing this forces our model to return the same Q values for all equivalent states.

2.3.3 Deep Q-Networks

After parsing the inputs, we need to estimate the Q-values. To do this we decided to use Deep Q-Networks. The concept of Deep Q-Networks was first published by DeepMind [2]. In the original paper the team used convolutional neural networks to parse the graphical outputs of Atari games and then used the Q-learning algorithm to make decisions. The team faced the problem, that deep learning is based on handling independent data, while making decisions influences future data. To solve this, they implemented an experience replay mechanism, a solution first proposed by Long-Ji Lin in his doctoral thesis [7].

Prioritized Experience Replay If we were to train our agent on transitions in the same order as it experiences them, it would result in a strong correlation between our fits. To avoid this, we build a buffer of replay data and sample from it. We use an adjusted approach called Prioritized Experience Replay [4] for this sampling, where, instead of sampling data at random, we select transitions in a way that favours those from which our model can learn a lot. The details of this are discussed in subsection 3.1.

Fixed Q-targets To further improve our model, we used two different networks, one to predict the value of the next state and another one to choose the best action. By splitting Q-value estimation (Q_{target}) and action selection (Q_{model}), we break the correlation between the weights that we are adjusting and the target values. The Q-values are then updated according to:

$$Q_{model}(s,a) = r + \gamma\, Q_{target}(s', \arg\max_{a'} Q_{model}(s', a')) \tag{5}$$

The weights of the evaluating network are kept fixed for a number of replays, and are then updated with the weights of the decision network.

Dueling DQN Since the Q-values represent the value of a state and the advantages of the possible actions from that state, our next refinement was splitting their estimation into two separate networks. This technique, called Dueling Deep Q Networks [3], splits these two aspects into two separate values:

$$Q(s,a) = A(s,a) + V(s) \tag{6}$$

Where $V(s)$ is the value of a given state s and $A(s,a)$ is the advantage of taking a given action a in the state s[1]. This enables the network to predict the value of a state without caring about actions, and determining advantageous actions without caring about the state value, which is useful since the features of a state that determine its value can be quite different from the features dictating the best next action.
However, we are now faced with a new problem. To arrive at the Q-values it would be incorrect to simply add $A(s,a)$ and $V(s)$:

$$Q(s,a) = V(s) + A(s,a) \tag{7}$$

While the equation in itself is correct, there are multiple $A(s,a)$ and $V(s)$ resulting in the same $Q(s,a)$. Thus, we would not be able to identify the correct $A(s,a)$ and $V(s)$ for a given $Q(s,a)$, which is a problem for our neural network, since it makes us unable to properly perform the back propagation.
We can solve this by one of two ways: The first solution is to force the $A(s,a)$ to at best be zero. This is done through adjusting the equation by subtracting the maximum advantage from the available actions:

$$Q(s,a) = V(s) + [A(s,a) - \max_{a'} A(s,a')] \tag{8}$$

Therefore, we are left with:

$$Q(s,a^*) = V(s) \tag{9}$$

Alternatively, we can replace the maximum of the advantages with their average:

$$Q(s,a) = V(s) + [A(s,a) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a')] \tag{10}$$

In the second version, the value and advantage functions lose their original meaning as they are now off-target by a constant. However, we still chose this version for our model, since it improves the stability of the optimization.

[1]Parameters θ, α, β are left out for the sake of readability, since they are not relevant to this discussion. The complete equations can be found in the original paper.[3]

2.4 State

After planning out the architecture we needed to create a state vector suitable to our approach. We used a stack of 17×17 matrices as our state vector. We encoded the different aspects of our state as one-hots. This considerably increased its total size, but was necessary for our model. If we would for example just use the arena matrix the game state provides, our network might think of crates as the opposite of walls, something that is not quite useful.

Our state then contained three matrices encoding whether there are walls, crates and bombs on a specific tile, with a one meaning yes and zero meaning no. Stacked on top of them are two layers for the agent position and whether it can use a bomb, the agent position being a one-hot of all arena positions, and the bomb indicator directly above the agent position. We further added four layers for the different states of bombs and two for explosions, so that e.g. one layer encodes whether there is a bomb with counter two at a particular position. Lastly, we added two more layers like the one for our agent positions and the bomb indicator, but this time for all other agents.

We split our training into three tasks. For the first task, only collecting coins, the state size can be reduced considerably, since only the three layers for walls, coins and the player position are needed, assuming bombs are excluded from the possible actions.

About 40 % of our state vector is static, since walls completely block a tile. However, our convolutional network still needs those tiles to be present, so we decided against excluding them.

2.5 Final Architecture

The implementation of all the principles discussed above resulted in the following architecture:
We decided to use Keras with TensorFlow as the back-end due to our familiarity with it. As shown in Figure 6, our state vector (here R was used, since this letter has no symmetries) is mirrored and rotated into the eight different equivalent versions. The resulting vectors are then passed into three consecutive convolutional layers[2] each with a ReLU activation function to parse the input layer as discussed in section 2.3.1. As described above our state vector is a stack of 17×17 matrices. While this looks 3 dimensional on the surface, each element of the stack is treated like the color channel of an image and can therefore be appropriately parsed by the Conv2D layers. After passing the convolutional layers the output is flattened.
We pass this flattened output into two different networks consisting of two Dense layers[3], one layer with a ReLU activation function and another one with a linear activation function, each. One network predicts the value of the state, the other one the advantages of taking specific actions. The resulting $V(R)$ and $A(R, a)$ are then reunified to form the Q-value $Q(R, a)$ according to Equation 10. Those

[2] Conv2D, the Keras version of a two-dimensional convolutional layer
[3] Keras version of a fully connected layer

are then permuted and averaged as discussed in subsubsection 2.3.2, resulting in the final Q-value.
The different parameters for the layers are given in Table 1, the final model has 721415 trainable parameters.
The model was compiled using Adam [8] as the optimizer with the mean squared error as the loss function. We chose Adam since it has been shown to quickly produce good results reliably.

Parameters for the different layers			
Convolutional layers			
	Filters	Kernel	Strides
Conv2D	128	4×4	1×1
Conv2D	64	4×4	2×2
Conv2D	64	3×3	1×1
Dense layers			
	Units		
Dense	256	Value	
Dense	1		
Dense	256	Advantage	
Dense	6		

Table 1: Parameters for the different layers in our model. The number of units in the last layer is the number of actions, six for the whole game and four for the first task.

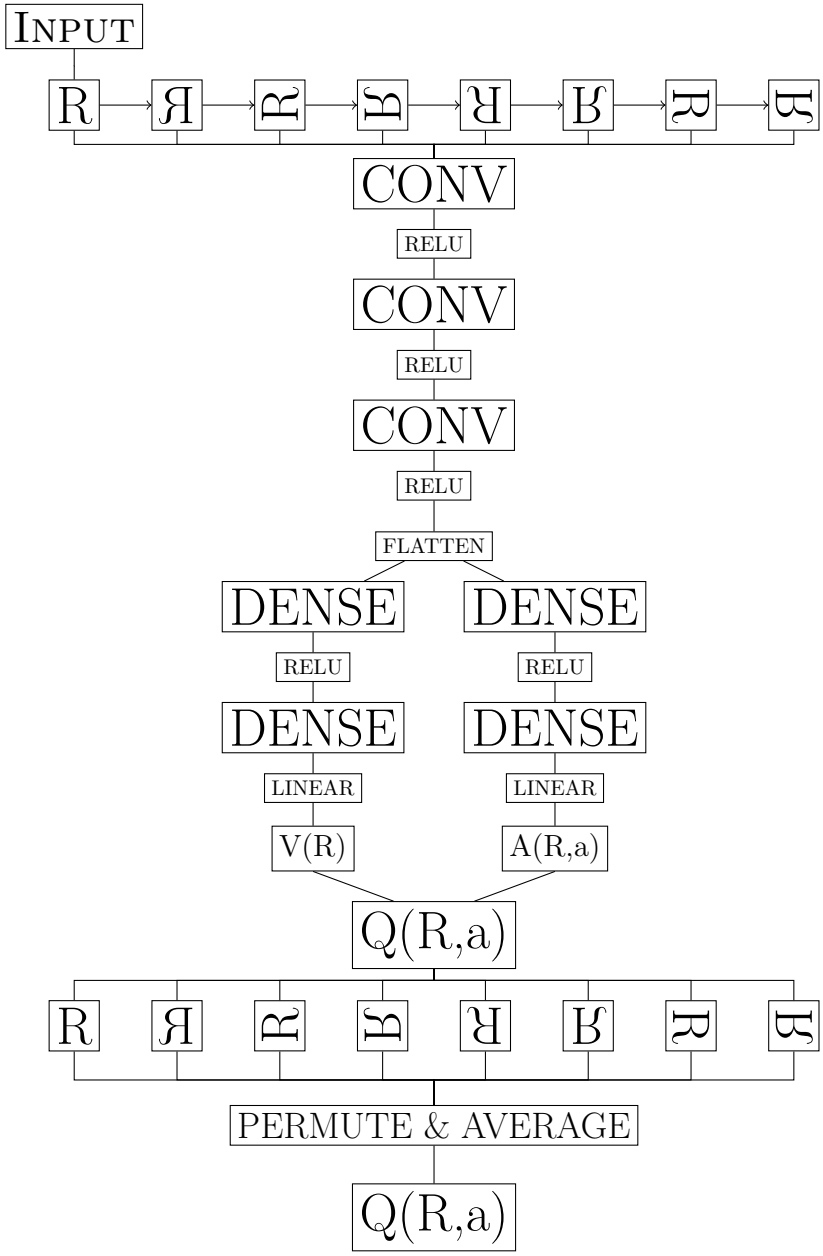

Figure 6: Model for symmetric duelling deep Q learning.

3 Training (Valentin)

3.1 PER

Experience replay, i.e. not only fitting with current transitions but also with older ones, is necessary for three reasons.

First, the transitions, as they are experienced, are highly correlated. To prevent this correlation from affecting the learning process, we should not fit transitions in the order that they are experienced. Second, we want the agent to learn the whole game. If we were to fit sequentially, it would for example fit for the beginning of the round in the beginning, resulting in worse performance at the end of the round. But because it would then fit for the end of the round, it would also perform worse at the beginning. So it would improve its behaviour only for the situations it is less likely to encounter next. Third, it is more efficient to fit with the same transitions multiple times.

To avoid those problems, we add every transition the agent experiences to a memory, and fit with batches of 32 transitions every four steps.

We do not fit with single transitions to achieve a more stable learning process. Our optimizer changes the weights in a way that optimizes the loss of the whole batch, which prevents them from being changed in a way that would be beneficial only for the current transition, and detrimental for most others. This way, the weights are updated in a way that results in a better performance for this larger sample, which is more likely to be representative of the whole game.

We could draw these batches randomly from our memory, however, most of the transitions are not particularly interesting. For example, in a round that may take hundreds of steps, only three agents can be killed. Therefore, most transitions would not contain those highly interesting events, and their probability of being used to fit the agent would be correspondingly small. To selectively use those transitions from which we can learn the most, we use prioritized experience replay [4] (PER), where we save transitions with a priority and draw them with a probability that is proportional to this priority.

We use the square of the absolute error between our prediction for a transition and the actual reward, called the mean squared error (MSE) δ, to calculate the priority p. To avoid sampling some transitions too frequently, we use an upper bound for δ, δ_{max}. Since transitions with error zero would have probability zero of being drawn from the memory, we also implement a lower bound, δ_{min}.

In the original paper, the authors used the temporal difference error for prioritization. However, since we use the MSE as the loss of our model, we should also use it for the prioritization.

To make sure that every step is shown to our model at least once, and since we do not know the error for new transitions, we add them to our memory with the maximal priority.

However, simply using the error as the priority would be prone to overfitting, since this would aggressively prioritize transitions with a large error. There would be a lack in diversity of the learning examples, and, since prediction errors fall slowly, transitions that by random chance have a larger error when

they are first encountered would be replayed frequently, while those that have a lower error would be ignored. To somewhat alleviate these problems, we use stochastic prioritization. The MSE is taken to the power of an exponent $\alpha \in [0, 1]$, with $\alpha = 0$ being equivalent to random sampling and $\alpha = 1$ to simply using the MSE for prioritization.

$$p_i = min(|\delta_i| + \delta_{min}\,,\, \delta_{max})^{\alpha}\,, \quad \alpha \in [0, 1] \tag{11}$$

This reduces the amount of prioritization in our sampling, leading to more stable learning.

Then, we sample transitions from our memory with a probability P that is proportional to this adjusted priority.

$$P_i = \frac{p_i}{\sum_k p_k} \tag{12}$$

Drawing from this distribution instead of a uniform one then leads to faster learning, since those transitions from which our agent can learn a lot are used for training more often. However, convergence of our model towards the true Q values requires that we draw the samples we use for fitting in the same way as they are experienced. To compensate for this, we use importance-sampling weights ω in our fit, with N being the capacity of our memory. We do not know why the authors of the original paper included N in this calculation, since normalizing our weights means it simply cancels out, and therefore did not include N in our implementation.

$$\omega_i = \left(\frac{1}{N} \cdot \frac{1}{P_i}\right)^{\beta}\,, \quad \beta \in [0, 1] \tag{13}$$

For stability reasons, the weights are normalized with the maximal weight, so that they can only scale the update downwards. For some reason the authors included the factor N in this calculation, but since we normalize the weights is should not make a difference.

$$\omega_i' = \omega_i \cdot (N \cdot P_{min})^{\beta}\,, \quad P_{min} = \frac{(\delta_{min})^{\alpha}}{\sum_k p_k} \tag{14}$$

Those weights are then used to weight the individual transitions when calculating the loss of our prediction.

$$MSE_{\text{loss}} = \sum_i \omega_i' \delta_i \tag{15}$$

For $\beta = 1$ this completely cancels the effects of importance sampling. We start with $\beta < 1$ at the beginning, and then linearly raise β to one over the course of training.

To save computing power, and since the learning rate is in general quite small, we used the prediction error before fitting for prioritization.

Implementation

To efficiently implement this structure, a sum-tree, as proposed by the authors of [4], was used.

We use a modified version of the implementation provided in [9]. Due to the limited time at our disposal, we decided against implementing this from scratch. However, we nevertheless spent enough time with it to completely understand the way it functions. Learning from the way others have implemented algorithms usually provides a steeper learning curve than trying to build everything from the ground up, and we wanted to learn as much as possible during this project.

Also, we spent enough time with this implementation to notice a grave error it, and all other implementations of this that we found (since they are mostly copies of this one), made. They all use the TD error to prioritize experience in their effort to slavishly follow [4], while using the MSE as the actual loss function of their model. The original paper only used the TD error because it was also used as their loss function. We corrected this, and, as was also done in [1], used the same error for prioritization as for our loss function, in our case the MSE. If this is not done, the model will learn slower, since the prioritized transitions are not necessarily those with the worst prediction, and most importantly will not converge, since the importance sampling weights only counteract the sampling method if the same error is used for sampling and fitting. Also, they used the batch size instead of the capacity for N when calculating the weights. However, we complete left this out as it should not make any difference.

On a related note, the same people did not seem to understand the idea behind double deep Q learning [10]. They often claimed to use it, while actually using only the fixed Q-targets proposed in [2] for deep Q-learning. Double deep Q-learning actually means training two independent models, instead of periodically updating the weights of a target model with the weights of the model that is trained.

All things considered, this supports the idea that simply copying the solutions other people wrote is in general a very bad idea.

The sum-tree is a binary tree with capacity N and consists of two arrays, one of length N containing the transitions, and one of length $2N-1$ containing the priorities. The priority of every node is the sum of the priorities of its (two) children. So the priority of the parent node is the total priority in the tree, the priority of the intermediate nodes is the total priority of their respective sub-tree, and the priorities in the last layer are the priorities of the transitions.

When an element is added to the tree, its priority is propagated upwards, i.e. added to all nodes above it. Likewise, changes in priority, either positive or negative, are added to all nodes above it. This ensures complexity $\mathcal{O}(log(N))$ for inserting elements and updating their priorities. New transitions are added to the right of the last element, starting from the left, and when the tree is full they are overwritten starting again from the beginning.

To sample from this tree, a random number p between zero and the total priority is drawn from a uniform distribution. The tree is then traversed downwards. If p is smaller than the total priority p_{left} of the left child of the parent

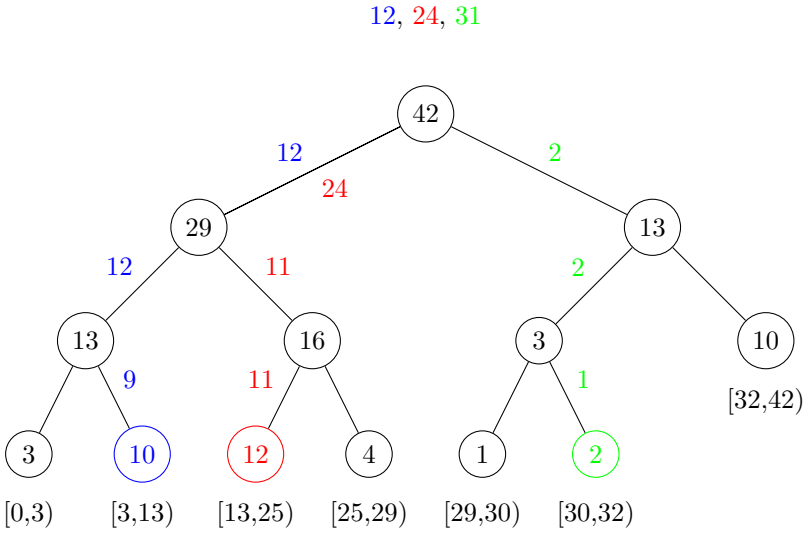

Figure 7: A sum-tree with capacity seven. Sampling for 12, 24 and 31.

node, searching continues in the left sub-tree. If it is larger, searching continues in the right sub-tree, and p is reduced by p_{left}. Then, the probability of choosing the left sub-tree is proportional to its share of the parent priority, and p is again a number drawn from a uniform distribution between p_{left} or p_{right}, respectively. This is continued downwards until the last layer is reached, and the element of the node that was reached is drawn from the tree. In this way, the probability of reaching a node is directly proportional to its priority, while also achieving complexity $\mathcal{O}(log(N))$ for sampling.

However, if all samples of one batch are sampled in this way, they could again be correlated. The p are random numbers, and can be arbitrarily close together for some batches. This would again introduce a correlation between our samples, since new elements are added next to each other. To completely avoid any such correlation, the batches are sampled in a slightly different way. First, the total priority of the tree is divided into k ranges, where k is the size of the batch. Then, k random numbers are drawn from uniform distributions in these ranges, and used to sample k elements. In the end, this does not change the absolute probability for an element to be drawn from the tree. However, it effectively divides the tree into sub-trees of equal total priority, and a sample is drawn from each of them. This ensures that the elements of our sampled batch are spread out over the whole memory, instead of possibly lying close together, and therefore that our agent learns in a way that is beneficial for the whole game.

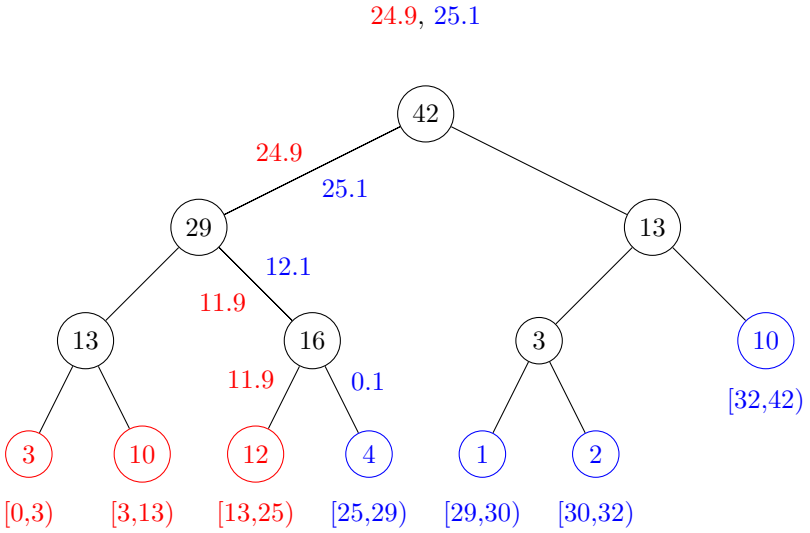

Figure 8: The same sum-tree as above. Separately sampling from the ranges [0,25) and [25,42] does not change the overall probability of elements to be chosen, but only draws elements from the respective sub-trees.

3.2 Exploration

For exploration, we use an ϵ-greedy policy with decreasing ϵ. We start out with $\epsilon = 1$, where epsilon is the share of actions that are taken randomly. ϵ then decays exponentially towards a minimal value ϵ_{min}.

$$\epsilon = e^{\Delta\epsilon \cdot N_{steps}} \tag{16}$$

However, the desired amount of exploration is dependent on the state. When there are bombs threatening the agent, a lower value of ϵ is appropriate than when there are no bombs in its vicinity, since a specific series of actions is needed to escape from a bomb and prevent the agent from dying. Therefore, we used two different values for ϵ, depending on whether there are bombs in the agent's proximity. Close bombs are detected by checking a rhombus with length and height 9, i.e. two times the bomb strength plus three, with the agent's position at its centre, for bombs. This rhombus then contains all bombs that could be a threat to the agent in the next step. In those cases, a different ϵ, ϵ_{bomb}, is used.

Additionally, breaking oscillations introduces further random actions, introducing another form of conditional exploration.

For choosing the random actions, we used relative probabilities. The probabilities of all impossible actions were then set to zero, and the remainder was normalized to one. This has the benefit of being slightly better than choosing for example walking with a fixed probability. In our approach, using bombs is more likely if the agent has not much room to move, which usually means there

are more crates and enemies in the proximity. For example, when the agent can move freely in any direction, the probability of choosing to use a bomb is only $\frac{4}{46} \approx 9\%$, while it is $\frac{4}{16} = 25\%$ when the agent can only move in one direction.

Relative Probabilities					
Left	Right	Up	Down	Wait	Bomb
10	10	10	10	2	4

Table 2: Relative probabilities used throughout our training process. The probabilities of waiting and using bombs were set to zero for the first task.

3.3 Oscillations

A large problem during training were oscillations between states.

The simplest is waiting (and choosing invalid actions) when there is nothing happening. Since this does not change the state at all, the model will again choose this as the next action, until it is interrupted by the environment. Since we train every few steps, it is theoretically possible that one of these updates to the weights would break this correlation, but it is not probable. However, if it is allowed to do this for hundreds of steps until the round ends, this leads to the memory quickly being filled with such bad actions. This behaviour can then get burned into our model, which is something we would like to prevent. So when this behaviour is detected, the next three actions are forced to be randomly chosen. Since waiting was determined to be useless, for the first of these random actions waiting is excluded from the possible actions. Such useless waiting is detected by checking whether the agent waited in a static situation, i.e. when no bombs are ticking and no explosions are present. To also function with other agents changing the state, whether the states before and after the action are equal is judged by comparing a rhombus with length and height 9 around the agent's position. This rhombus encompasses all bombs that could threaten the agent, so that for example waiting before a bomb explodes to not walk into the explosion is not prohibited.

A slightly longer oscillation is alternating between two states. Again, the model is not likely to break this cycle by itself. So we again force the agent to take three random actions if this behaviour is encountered. This is detected by comparing the second last state and the state after the current action. A step was useless if the same rhombus as above around the next and second last state are the same, and the agent is at the same position as before.

The same can be done for oscillations of arbitrary length, however all oscillations shorter than eight steps are already covered by this. The shortest oscillation that does not involve stepping back on the field the agent came from is walking around a wall, which takes eight steps. So we checked for oscillations of length eight to thirty steps and forced eight random actions afterwards. Here, we compare the whole state, e.g. eight states ago, with the state after the current action. One could theoretically still implement a version that com-

pares only a local part of the whole state, however, the oscillations we observed were quite elaborate. They might encompass twenty-five steps and three bombs, which means that the part of the state that would need to be checked is a large proportion of the whole state vector. We therefore decided to just compare the whole state at once. However, using the rhombus used above and extending it around all positions visited by the agent would improve this detection

All of the above is only relevant if none of the actions in the loop were taken randomly, so we only forced random actions if all of them were decided upon by the agent.

3.4 Rewards

The reward function assigns a value to every event that is given to the agent by the environment, and additionally auxiliary rewards were defined.

We could punish all actions with a small negative reward so that our agent learns to be efficient. However, this is redundant, since fitting for the discounted reward already ensures that the agent will prefer efficient paths to rewards. There is a negative reward for impossible actions, however, they are already excluded from the possible actions our agent can take.

Destroying crates is associated with a positive reward. This reward needs to be chosen large enough that the agent can improve its expected reward by only destroying crates. Otherwise it quickly decides that a swifter death is preferable and learns to commit suicide. Since, when choosing random actions, the agent does not live long enough to experience dying enemies, and collecting coins does not happen very often, a smaller reward for destroying crates can not be compensated for by increasing their rewards.

The reward for collecting coins should be chosen large enough that the agent prefers collecting coins to destroying crates, which is the better strategy since our agent is not awarded points for destroying crates. We decided against a reward for finding coins under crates. On the one hand, this would facilitate searching for coins under crates. But on the other hand, this can also be achieved by raising the reward for destroying crates, and without introducing randomness to the reward function due to the coins being randomly distributed. So we chose to keep our reward function as deterministic as possible.

Dying is punished with a large negative reward, with the reward for getting killed slightly larger than the reward for suicide, since getting killed rewards the other agent with points.

Killing agents is rewarded with a large positive reward, ideally somewhere around five times the reward for collecting coins. There is also slightly smaller reward if agents were killed by other agents. This is not necessarily correlated with the actions of our agent, but happens comparatively often. Since killing agents does not happen a lot while our agent is still struggling, we wanted the agent to learn as soon as possible that it profits when other players die. Also, the game state does not contain information about the creator of specific bombs, meaning our game state did not contain it either. Therefore, differentiating between our agent killing enemies and other agents killing them should anyway

be quite hard for the model. However, even if the rewards would be equal, our agent should still learn to work towards their death, since we use the discounted reward for fitting.

Since the state vector does not contain the number of steps already taken, and since we do not need the agent to learn the maximum length of one round, we chose not to reward surviving the round. The agent should anyway learn to survive as long as possible as a means of avoiding dying. Likewise, we only chose to fit a state as having no next state, i.e. the reward to fit for is only the reward of this step, when the round ended before the maximal amount of steps. This can happen either because our agent died, or because it is the only surviving agent in an empty arena, the only states where the agent should expect the round to end.

There are separate additional rewards for actions that result in an oscillation to discourage the agent from getting stuck.

Using a bomb without destroying crates or other players also gets punished to further encourage using them efficiently.

Waiting when in direct range of a ticking bomb is similarly punished. The calculation whether the agent is threatened is sensitive to possible walls protecting it.

There is a negative reward for getting stuck behind bombs. This is the case if the only valid action the agent can take is to wait, here walking into explosions is considered a valid action since they could fade and still allow the agent to escape. This is meant to make the reward less sparse, as dying from a misplaced bomb only happens four steps after the action was chosen. This reward will also be triggered if a player blocks its movement, but this will not happen very often, and if it does, it would still be a dangerous situation. It is only applied for the action that led to the agent being stuck, not for those that follow it.

The rewards for the final agent are summarized in Table 3.

Rewards for the final agent	
Executed actions	
Walked (left, right, up down)	0
Waited	0
Impossible action	- 50
Bomb dropped	0
Events	
Bomb exploded	0
Crate destroyed	100
Coin found	0
Coin collected	1000
Killed opponent	5000
Killed self	- 1000
Got killed	- 2000
Opponent eliminated	3000
Survived round	0
Auxiliary rewards	
Oscillation of length 1 (wait)	- 100
Oscillation of length 2 (alternate)	- 100
Oscillation of length ≥ 8 (elaborate)	- 100
Useless bomb	- 50
Waited in range of a bomb	- 100
Stuck behind bomb	- 100

Table 3: Rewards that were used to train the final agent. They can be used to train an agent for the different tasks as well.

3.5 Choosing Actions

Depending on the state vector, not all actions are possible. For example, walking is only allowed in directions where there are no crates, walls or bombs. In principle, our agent is able to learn which actions are valid, assuming a negative reward for invalid actions. However, there does not seem to be a benefit in learning this. Invalid actions are equivalent to waiting, meaning their effect can always be replicated just as easily with valid actions, and about one half to one third of all actions from any state are invalid. Therefore, allowing them and starting with random actions would mean a large amount of the transitions in our memory would consist of invalid actions, i.e. waiting, resulting in slower learning of the strategies we are actually interested in. So we decided to just prevent our agent from taking invalid actions in the first place. To include this in the learning process, the action taken when fitting for the next step after a transition is also subject to those constraints.

For every direction, we evaluate whether there are obstacles (walls, crates,

bombs) preventing it from walking in that direction. Using a bomb is only possible if the game says it is, waiting is possible unless the agent is standing on a bomb. We made sure that waiting is possible if all other actions are forbidden.

We also prevent the agent from walking into explosions. Here, the condition for an explosion is that there would be an explosion on the field the agent steps on when it steps on it, i.e. explosions and bombs are projected one step into the future. Whether an explosion is in the way is then evaluated with the future state. This prevents all deaths where the last action could have saved the agent.

To alleviate the impact all these computations have on the performance of our agent, we used vectorization wherever possible. This is especially important when determining the possible actions, since this also needs to be done every time we train our agent. For the same reason, the computation accepts batches of arbitrary size.

3.6 Training Strategy

To allow for convergence of our Q values, we used a decreasing learning rate. It falls of as:

$$\eta = \eta_0 \cdot \frac{1}{(1 + \Delta\eta \cdot N_{replay})} \tag{17}$$

Here, N_{replay} is the number of replays.

The target model weights are always updated every 200 fits, except for the last iteration of training for task three, where we updated them every 2000 fits.

The parameter β was raised by an increment every time we replayed a batch of transitions.

All training was done with a maximum of 400 steps per round, if the agent was still alive at that point the round was aborted and the next one started.

With the exception of the agent for task one, the rewards as specified in Table 3 were used.

Task 1 (Coins)

For this task, we only allowed walking, while excluding dropping bombs and waiting, since they are not necessary for only collecting coins. Then, we could use a significantly smaller version of our state vector, containing only information about walls, coins and the agent position.

We let the agent choose random actions for enough rounds to fill the memory with experienced transitions. Then, we started training our agent by replaying batches of 32 transitions every four steps, while simultaneously reducing the learning rate and the exploration rate ϵ. Since there were no bombs yet, we only used a single exploration rate.

We prevented the agent from choosing invalid actions, meaning it only had to learn the most rewarding way of walking around the arena, which is collecting coins as fast as possible. Also, we only had to specify a reward for walking,

collecting coins and getting stuck in oscillations, since those are the only relevant events the agent encounters.

With this setup, the agent performed almost as well as the provided simple agent after about six hours of training on a GPU.

This training was done while there were still some quirks in our code. After fixing them, we achieved a performance for the third task that was far superior than that of the simple agent. So we probably could have achieved a better performance here, had we trained the improved agent on the first task again. Also, we were too conservative with the learning rate at the beginning. So using a larger learning rate for this task would probably improve the training.

The parameters can be found in Table 4.

Task 2 (Crates)

The state vector used for this task was the full state that also contains information about other players, since we wanted to continue training from this baseline instead of learning everything from scratch for the third task.

We again let the agent choose random actions for enough rounds to fill the memory with experienced transitions. Then, we started training our agent by replaying batches of 32 transitions every four steps, while simultaneously reducing the learning rate and the exploration rates ϵ and ϵ_{bomb}.

The trained model was still not perfect, continued training with a smaller learning rate could probably further improve its performance. However, by then we were running out of time and chose to focus on the third task instead, since the model was still good enough to use as an initialization for task three. The model seemed to get worse during the end of training, so we used the version after 10000 rounds for the next task.

In total, we trained the model for about 16 hours on this task. The version we used for the next task was a snapshot after about 10 hours of training time.

The parameters can be found in Table 4.

Task 3 (Full game)

When training our agent for the full task, we used the best model from task two as an initialization. In this way, the agent did not have to learn everything from scratch.

Here, we prohibited our agent from walking into explosions, and used decreasing exploration rates, a decreasing learning rate and experience replay as above.

We then trained the best model from task two against three simple agents.

Afterwards, we again used this model and continued training with a smaller learning rate and the same setup.

The first iteration trained for about 20 hours, while the final model we used from the second training was trained for about eight hours. So in total, the final model was trained for about 40 hours on a mid-range GPU.

The parameters can be found in Table 4.

From this final version of our model, we would have continued training with a form of self-play, e.g. training with two of our agents and two simple agents. However, we could not train our model against itself due to memory constraints. Loading one model for use with the GPU already fills most of the available VRAM of our graphics card, so we could not load a second model to play against, and training without using the GPU takes too long to be practical. We also did not manage to only let only one model use the GPU, since we could not find out how this could be achieved when using keras and tensorflow.

However, we managed to let multiple instances of our agent compete against each other when running on the CPU.

Training parameters for task one, two and three

	one	two	three 1	three 2
γ	0.97	0.97	0.97	0.97
η	0.001	0.0005	0.0001	0.00004
$\Delta\eta$	$\frac{1}{2560}$	$\frac{1}{70000}$	$\frac{1}{60000}$	$\frac{1}{100000}$
ϵ	1	1	0.1	0.05
ϵ_{min}	0.01	0.1	0.05	0.01
$\Delta\epsilon$	$-\frac{1}{40000}$	$-\frac{1}{30000}$	$-\frac{1}{300000}$	$-\frac{1}{500000}$
ϵ_{bomb}		1	0.02	0.01
$\epsilon_{bomb_{min}}$		0.02	0.01	0.001
$\Delta\epsilon_{bomb}$		$-\frac{1}{20000}$	$-\frac{1}{100000}$	$-\frac{1}{500000}$
Memory size	80000	80000	80000	80000
δ_{min}	2	1	1	1
δ_{max}	50	100	100	100
α	0.6	0.6	0.6	0.6
β	0.4	0.4	0.4	0.4
β increment	$\frac{3}{160000}$	$\frac{3}{2000000}$	$\frac{3}{2000000}$	$\frac{3}{2000000}$

Table 4: The training parameters we used to train the agents for task one, two and three.

4 Results (Tatjana)

The following chapter discusses the performance of the final version of our agent and also of the agent in its earlier development stages on the preliminary tasks.

We start with presentation of different development stages of the model. subsection 4.1 presents the model, which is trained only for the task of navigating a board ("safe" environment without bombs) and compare it with a random baseline and the simple agent. Then, we train the agent for the second task ("dangerous" environment with bombs but without enemies) in subsection 4.2. Training and performance of the final version of our agent for the full game is described in subsection 4.3.

Versions of our agent configured to load the trained models we discuss here,

together with all plots concerning their performance, can be found in our GitHub repository.

4.1 Task 1 (Coins)

We first trained a constrained model for the first task. It only has to navigate the board efficiently, without caring about waiting and using bombs.

The trained model is almost as good as the simple agent (Figure 9), on average it needs about 57.5 steps to collect all coins, while the simple agent needs only 55.

The model still sometimes gets stuck in oscillations, those are the rounds where it needs a large number of steps to complete the task. Further training could probably have improved its performance beyond that of the simple agent.

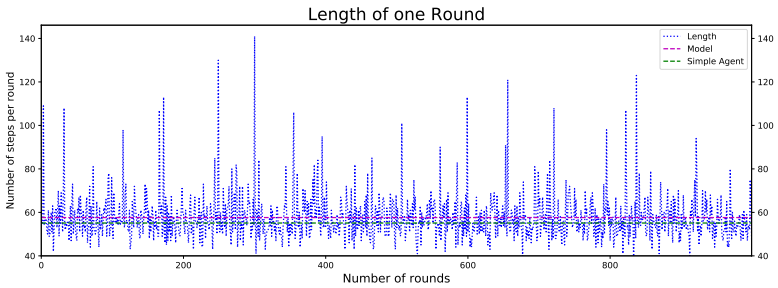

Figure 9: Task 1. Number of steps needed to collect all coins.

4.2 Task 2 (Crates)

For the second task, the agent must drop bombs to destroy the crates, find and collect all hidden coins and not forget efficient navigation on a game board. The vertical line in the plots always shows the start of learning, the rounds before this are used to fill up the memory.

Figure 10 shows the total reward per round achieved by our agent. One can clearly see the increasing improvement over taking random actions as our model learns. One has to remember that an increasing reward in itself is not sufficient, since agents tend to game their reward function if it is misspecified. However, an increasing total reward still shows that the agent is learning something. But the reward is nevertheless only a proxy, and the actual performance in terms of collected coins etc. is a better metric for successful learning.

All plots are shown over the number of rounds, however since we train every four steps, training for one round takes longer the better our agent is, i.e. the longer it lives without killing itself.

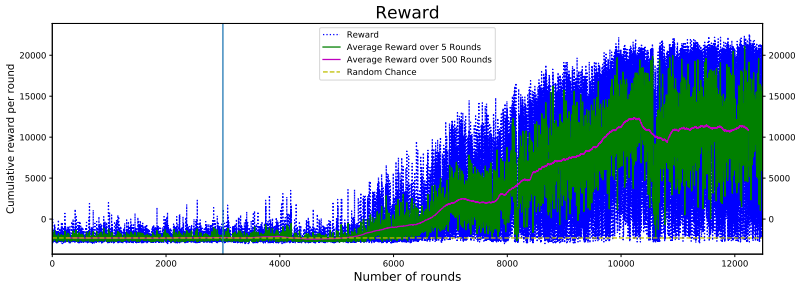

Figure 10: Task 2. Reward.

The number of crates our agent destroys (Figure 11) rises in step with the reward, or rather the reward raises with it. The number of coins it collects (Figure 12) rises in the same way, and the share of coins that is collected is approximately the same as the share of crates that is destroyed. This should not be taken for granted, but only occurs when the reward for collecting coins is sufficiently higher than the reward for destroying crates. If it is not, the agent will prefer destroying crates, and not collect all exposed coins.

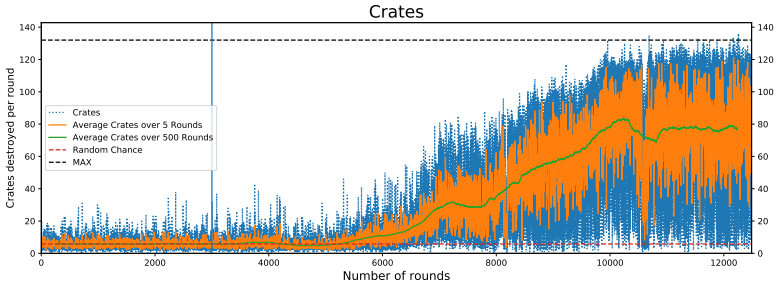

Figure 11: Task 2. Destroyed crates per round.

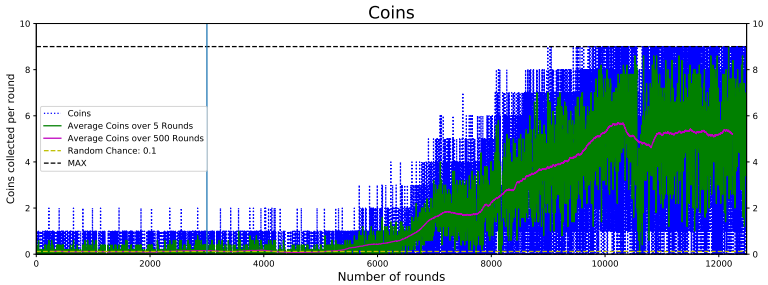

Figure 12: Task 2. Collected coins per round.

Figure 13 Shows hot the parameters of training changed over time. The curves do not look like the functions we use to adjust the values, this is again due to everything adjusting according to the number of steps, not the number of round. So e.g. the learning rate seems to fall faster once the agent gets better, but this is simply because we train the agent more often per round, and it is adjusted every time we replay a batch of transitions from our memory. ϵ_{eff} is the effective exploration rate, i.e. the total share of actions that was taken randomly. It can be larger than the normal exploration rate since we force random actions when the agent gets stuck in oscillations.

Technically, we should have trained until β reaches one. However, since we saw no more improvement after about 10000 rounds, we stopped training prematurely.

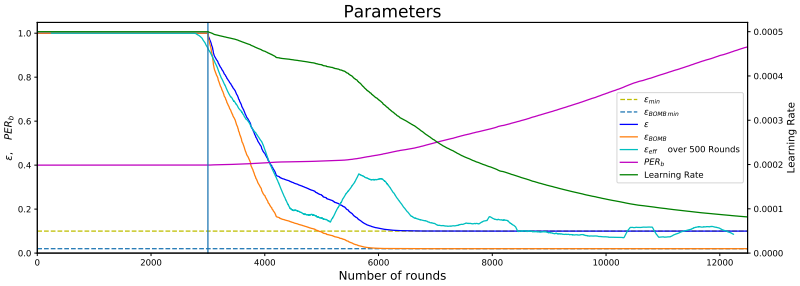

Figure 13: Task 2. Parameters.

The share of each action is shown in Figure 14. When the agent takes random actions, the rounds are relatively short. In the beginning of a round, it has to wait a lot since it does not have a lot of tiles that are not threatened by explosions, and also the random actions can not step on them. However, the share of waiting should fall as rounds get longer, as it does. Interestingly, the share of bombs that are used by random change is almost the same as the one used by the trained model.

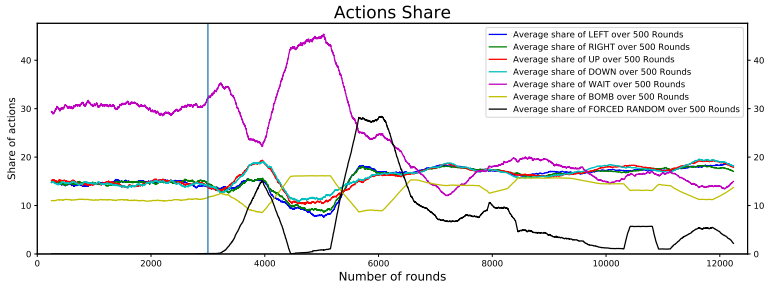

Figure 14: Task 2. Share of executed actions.

Overall, training was relatively stable. Every once in a while, our agent tends to get stuck in oscillations and the share of forced actions raises. However, since we prevent this behaviour from getting burning into our model, it always learns to overcome it.

Figure 15 and Figure 16 show the performance of the trained model without exploration, i.e. with ϵ and ϵ_{bomb} equal to zero. Here, we took the model after 10000 rounds instead of the final model, since it seemed to perform best. This model collects about 7.5 coins per round. This is still not optimal, but we did not want to spend even more time on this task. After all, the competition concerns the final task, and not the preliminary ones. This version was good enough to use as a starting point for the next task, since it was very good at the beginning of rounds. It apparently had problems identifying the last remaining crates. But this is not really relevant for the full game, since the agent encounters enemies well before this point is reached.

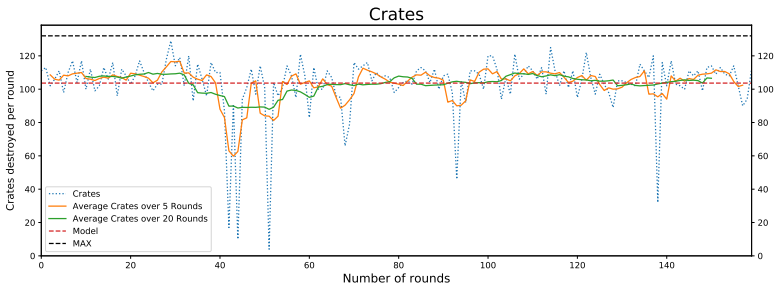

Figure 15: Task 2. Destroyed crates per round of the trained model.

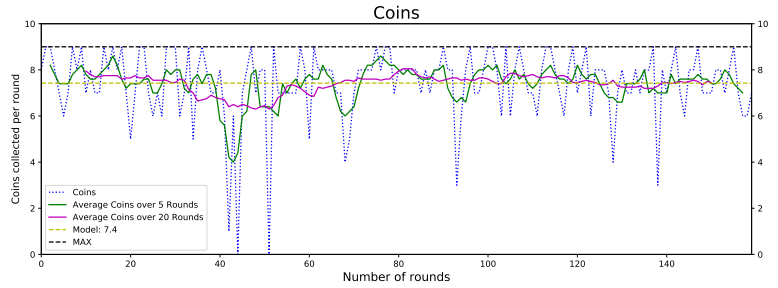

Figure 16: Task 2. Collected coins per round of the trained model.

The simple agent needs about 350 steps to collect all coins and destroy all crates. Since our model never cleared the whole board, we did not explicitly compare them, this would only be useful if we achieved a comparable performance.

4.3 Task 3 (Full game)

On the third task, the agent has to navigate a board with crates, drop bombs, gather coins and play against other agent for the highest score.

4.3.1 First iteration

We continued training from the model in subsection 4.2, which is the baseline in our plots. Figure 17 shows a slow improvement over time. There is a small dent after about 2000 rounds of training. This is probably due to the agent learning to engage enemies, which caused it to be killed more often (Figure 20). The performance improved again after it learned to avoid getting killed. However, Figure 19 shows that it did not do so by avoiding enemies, but by improving its behaviour such that it emerged victorious from those encounters more often.

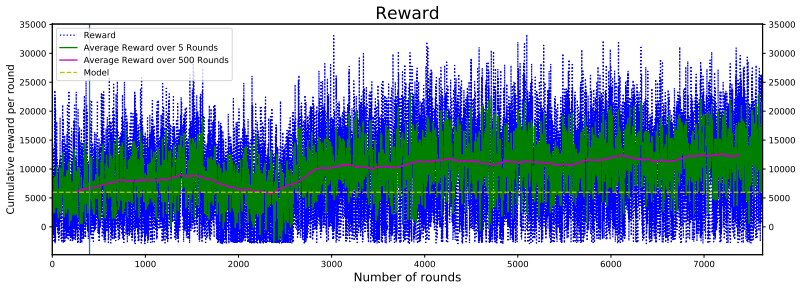

Figure 17: Task 3. Reward.

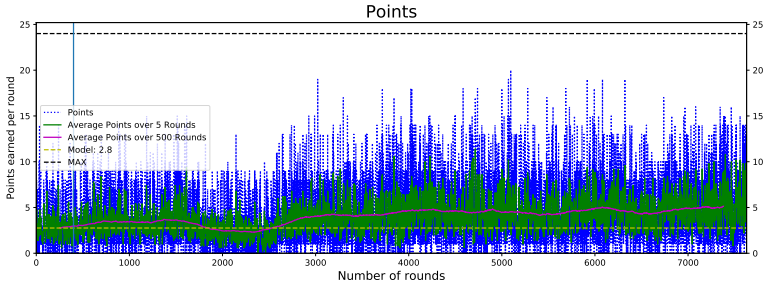

Figure 18: Task 3. Awarded points per round.

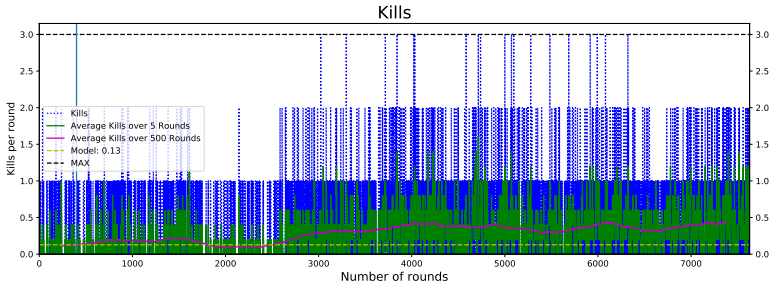

Figure 19: Task 3. Enemies our agent killed per round.

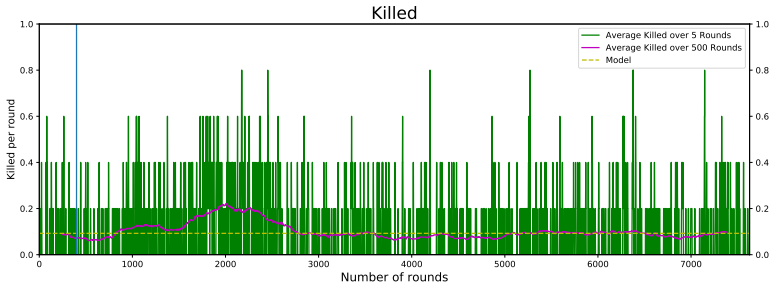

Figure 20: Task 3. How often our agent was killed by enemies per round.

The points it was awarded per round (Figure 18) and the number of coins it collected (Figure 21) followed this development. Interestingly, the number of crates it destroyed (Figure 22) and the amount of coins it collected stayed almost the same during training, although the number of coins improved from about two to about three per round. This shows that even an agent that ignores the other players can have limited success in this game. However, the number of points the agent was awarded improved more strongly, since killing other agents is five times as valuable as collecting coins.

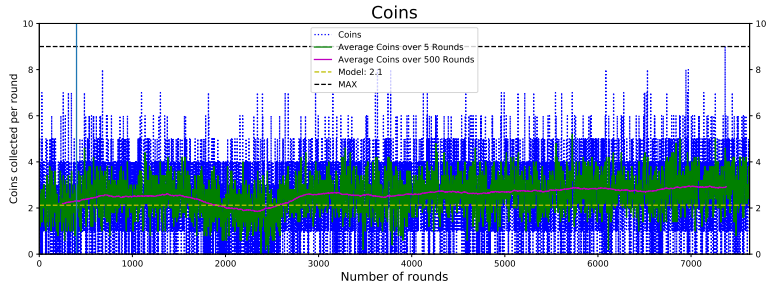

Figure 21: Task 3. Collected coins per round.

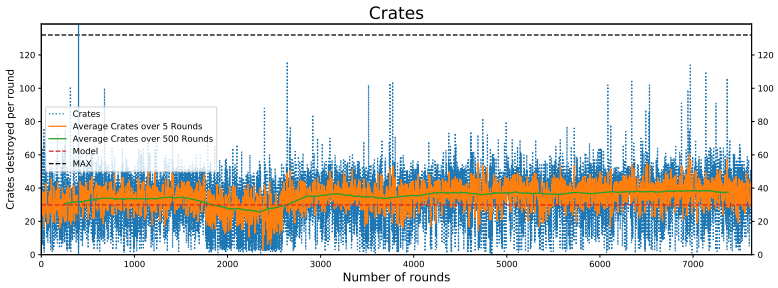

Figure 22: Task 3. Destroyed crates per round.

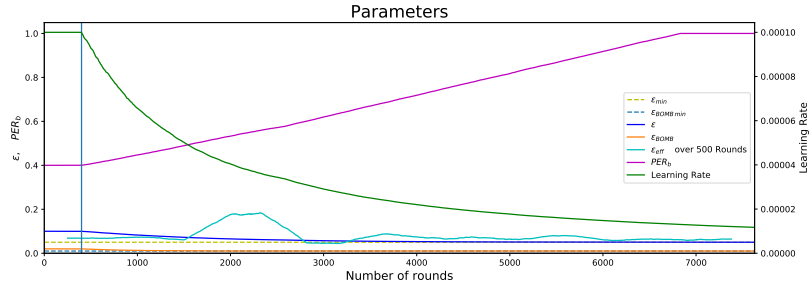

Figure 23: Task 3. Parameters.

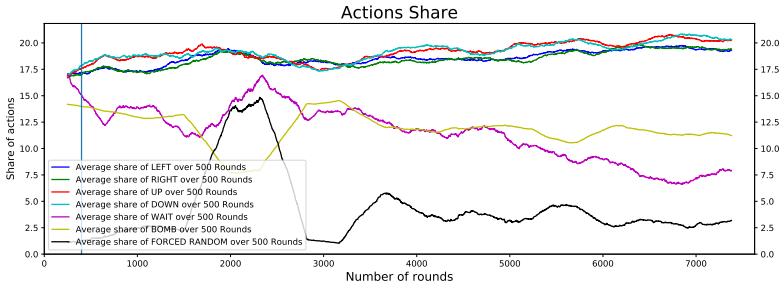

Figure 24: Task 3. Share of executed actions.

In the full game, we found it quite hard to judge the performance of our agent based on these plots alone. For example, only killing an enemy every two rounds seemed like an underwhelming performance of our agent. However, we then ran the game again to evaluate our agent (again without exploration), and this time we included a version of the simple agent that also exported the performance metrics our own agent supplied. In this way, we could directly compare our own model with the simple agents it played against. For our training, we stopped rounds from running after our own agent was killed. For this evaluation, we let every round continue until it was finished. Also, we set the punishment for being the slowest agent to zero. This would have reduced the mean points of our agent by one, since it took a few times as long to choose actions as the simple agents. However, we wanted a measure for the tournament performance of our agent, and there our agent will not necessarily be the slowest, so we excluded this.

To our surprise, doing so showed that our agent was clearly superior to the simple agent. It was rewarded twice as many points per round as the simple agents it played against (Figure 25). It killed two and a half times as many enemies as the simple agent did (Figure 26), while getting killed about half as often (Figure 27) and collecting about one and a half coins per round more (Figure 28). Overall, this a very good performance, supporting the assertion in the assignment that the simple agent plays the game only "reasonably well".

This also shows that killing agents is quite rare, since on average only one agent per round dies in a way the environment counts as a kill for another agent, while all nine coins are collected at some point in almost all rounds.

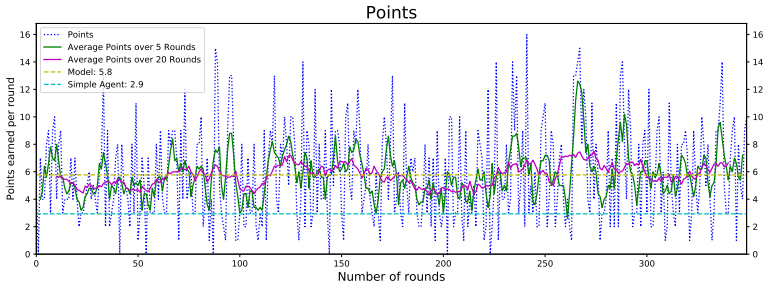

Figure 25: Task 3. Awarded points per round.

Figure 26: Task 3. Enemies our agent killed per round.

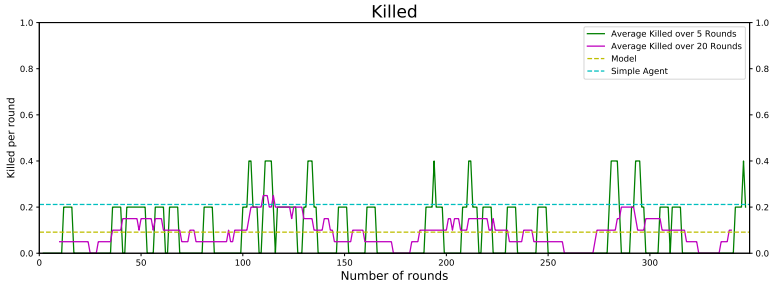

Figure 27: Task 3. How often our agent was killed by enemies per round.

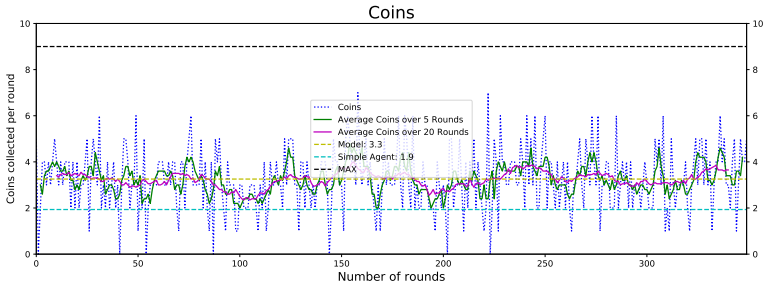

Figure 28: Task 3. Collected coins per round.

4.3.2 Second iteration

We then continued training the model. The improvement compared to the first iteration was quite small, and in the end the model actually got worse again.

However, the reward the final model achieved was as good as that of the best snapshot (Figure 29), this is probably due to the agent gaming our reward function. The metric we actually care about, the number of awarded points per round (Figure 30), is worse than that of the best snapshot, while the reward is about the same.

We therefore used the best snapshot, at 2200 rounds, as our final agent.

This need to take the best snapshot is somewhat disheartening, since it shows that our agent did not converge perfectly. However, even DeepMind talks about using "the agent snapshot that obtained the highest score during training" ([1]), so this seems to be a general problem of reinforcement learning.

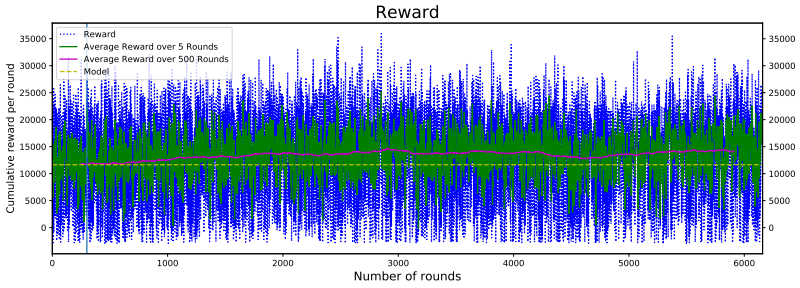

Figure 29: Task 3. Reward.

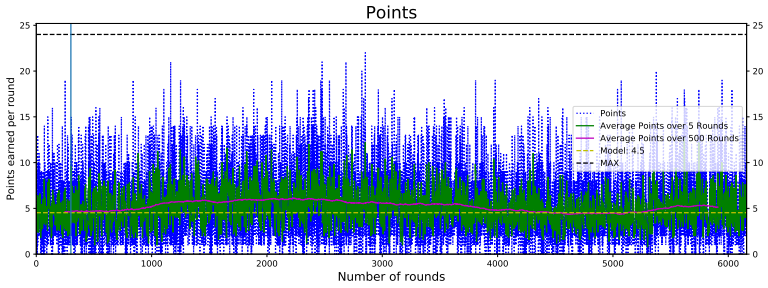

Figure 30: Task 3. Awarded points per round.

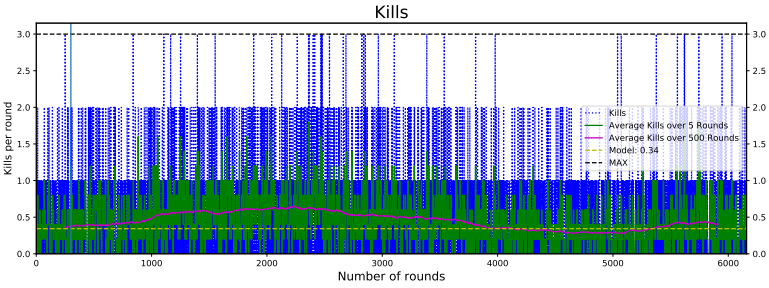

Figure 31: Task 3. Enemies our agent killed per round.

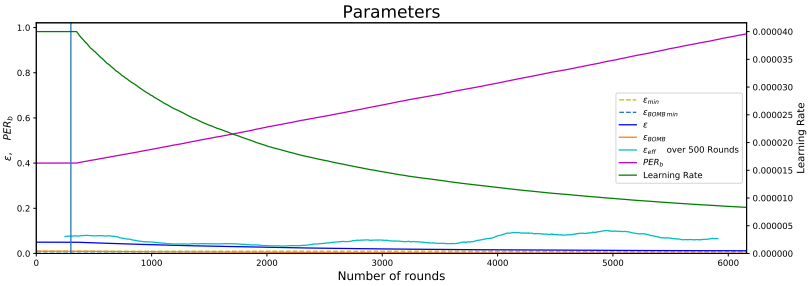

Figure 32: Task 3. Parameters.

We then proceeded to evaluate this model.

It turned out to be a small improvement over the previous iteration. It is awarded about half a point more per round (Figure 33), while now killing about three and a half times as many enemies as the simple agent (Figure 34) and still getting killed about half as often (Figure 35). It was slightly worse at collecting coins than the previous version (Figure 36).

However, we still used this as the final version since we only care about the total points in the competition.

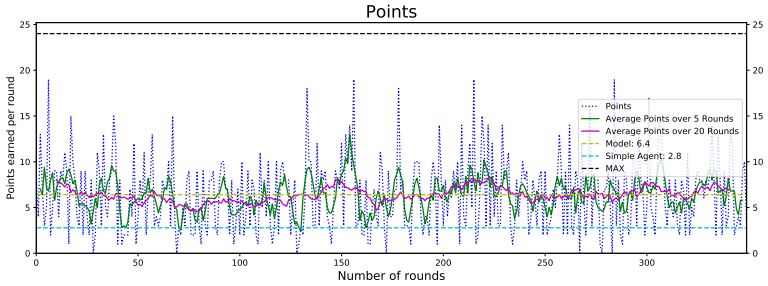

Figure 33: Task 3. Awarded points per round.

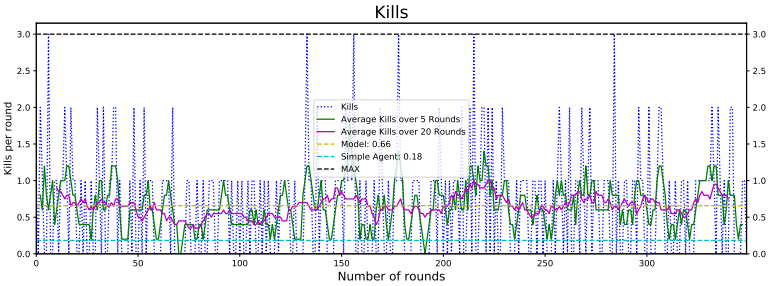

Figure 34: Task 3. Enemies our agent killed per round.

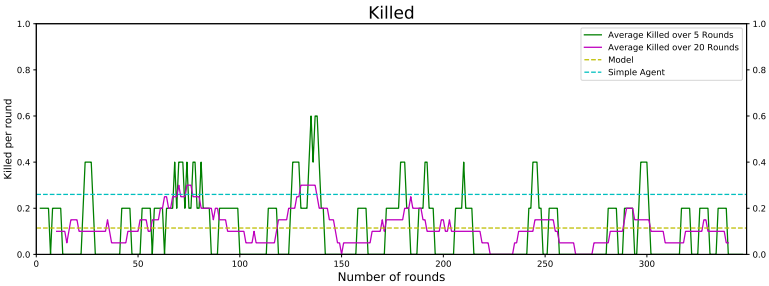

Figure 35: Task 3. How often our agent was killed by enemies per round.

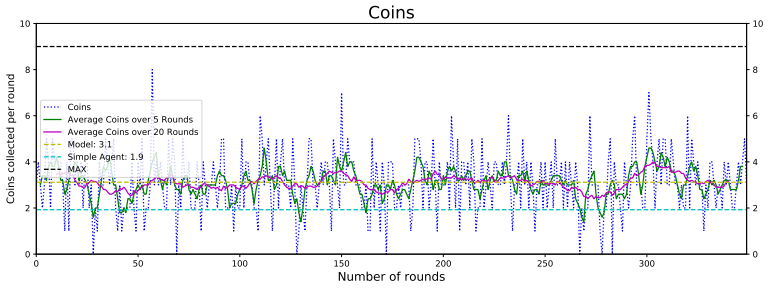

Figure 36: Task 3. Collected coins per round.

We also evaluated this agent on the different sub-tasks. Its performance for task one was significantly worse than that of the specialized model, failing to consistently collect all coins before killing itself or before the round ended (Figure 37).

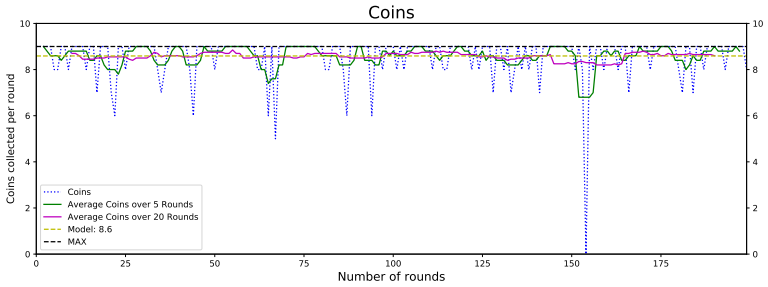

Figure 37: Task 1. Collected coins per round.

Likewise, it performed slightly worse than the agent it was trained from on the second task, collecting fewer coins (Figure 38) and destroying fewer crates (Figure 39).

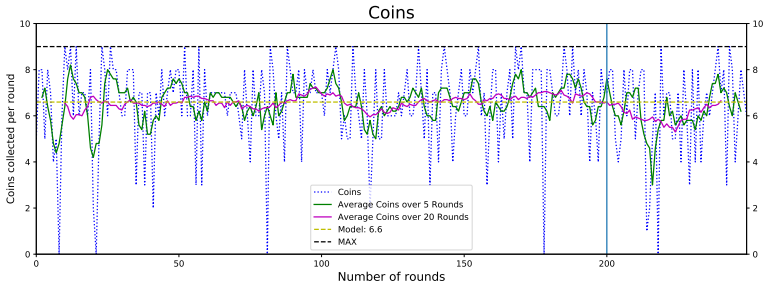

Figure 38: Task 2. Collected coins per round of the trained model.

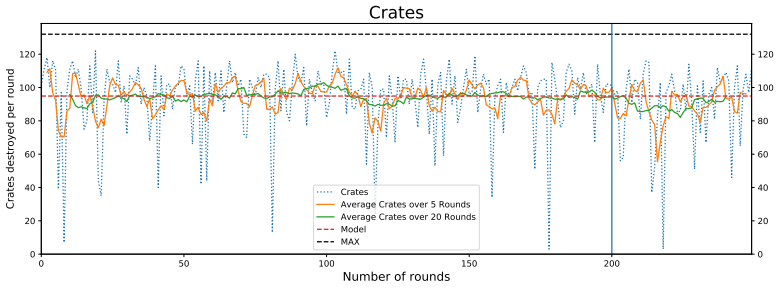

Figure 39: Task 2. Destroyed crates per round of the trained model.

But this performance was to be expected, since we never explicitly trained this agent on the different sub-tasks. And after all, the competition is about our agent killing other agents, not about collecting coins on its own.

Lastly, we made four instances of our final model compete against each other. They succeeded in killing each other (Figure 40), which shows that the strategies our agent learned generalize to other agents than the simple agent.

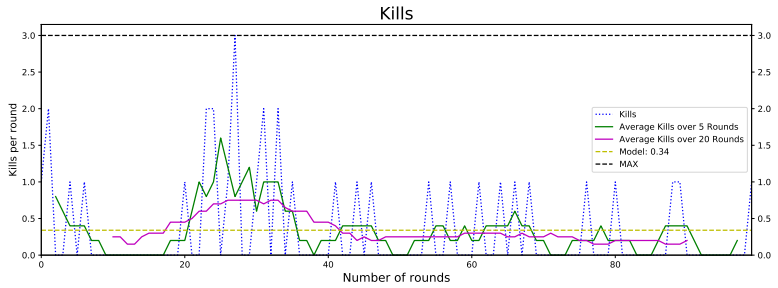

Figure 40: Task 3. Enemies our agent killed per round.

The final model needs about 30 ms per step of thinking time on our CPU, for the most part this is used for predicting actions, a smaller part is used for the computations necessary for releasing it from oscillations. This was done in the reward update functions during training, but since those functions are not called in the competition, we had to do it after acting.

5 Difficulties (Tatjana)

5.1 Cost of computations

One of the difficulties was a high cost of computations.

Our first idea was to use Q-Learning with a Q-table as an architectural solution. However, the size of the necessary Q-table made this solution impractical,

and we decided to use deep reinforcement learning.

What made the computations particularly time-consuming was the need to express the state as a One-Hot vector. This considerably increased its size, and therefore also the time it took to train our model.

5.2 Experiments and evaluation

The perfect solution needs a lot of experiments to evaluate the performance of the current model on a number of tasks, compare the learned behaviour of different agents and select an optimal decision by every improvement state. A strict scientific approach sifts out unsuccessful ideas and controls evolution of the model, instead of using trial and error method and guessing the right decisions. Practical conception with clear overview of the next steps allows to get a positive or negative feedback and understand that the chosen way is perspective (or not). However, training of the models returns appropriate results only if it is done on GPUs, and even using GPUs for training needs a lot of time resources. Thus, performing a sufficient number of experiments within the allotted time was not possible.

Tuning parameters of the model by its development stages and tuning parameters of the final model are tasks that have a strong impact on the results. A set of such systematic experiments was also not possible and could probably improve the performance of our agent. So, it is hard to guess whether our agent could learn a better policy with more time and computational resources, though the problem of finding an "approximately optimal solution" is usually a difficult one [11].

Evaluating the data is also not a trivial task. An algorithm tries to solve a general problem, and a good reward at a particular state does not mean that we get the same rewards in the same situation in a real environment. So, it is not smart to trust any particular rewards too much.

5.3 Preventing the agent from killing itself

This difficulty is also related to the problem of time-consuming computations. The possible solution would be searching a few actions in the future to exclude those that result in suicide. However, this would mean creating hundreds of possible future states and evaluating every one of them, something that would take about one second per step even with the efficient, vectorized approach we tried. Most importantly, this would also have to be done for every state with which we fit our model, blowing up the training time beyond what we could tolerate.

5.4 Breaking oscillations

One of the big problems was breaking oscillations, which was already discussed in subsection 3.3. Specific oscillatory patterns occur while training, and the model cannot break such cycles itself, choosing the same actions again and

again, until it is interrupted by the environment. Learning such oscillations is extremely undesirable, and breaking this cycles was a big difficulty. We tried to detect some of these patterns and construct behaviours to address these problems.

5.5 Exploration vs. Exploitation

It is very difficult to decide, if the agent has to follow the already learned policy or to take some new actions in hope that it would lead to a larger playoff. We tried to solve this question by exponentially decaying epsilon value with training time. That makes the agent to make more exploitation as it learns the policy. However, such answer is always a trade-off, because the chance of better actions always exists. Thus, we still keep the value of epsilon on a low level to keep the possibility of random behaviour and further evolution.

Such solution has not been enough. Opportunity of performing random actions is good in a safe environment, for example, on a first task, where the agent learns to navigate a board and does not face a problem of being burnt on a bomb. Bombs and enemies make the environment dangerous, and following the learned policy with small opportunity of making random actions is more preferable. Thus, another espilon value for "dangerous cases" was incorporated.

5.6 Difficulties by solving complex tasks

Learning of the full game is a task with a set of problems that must be handled by the agent at the same time. The agent must solve efficient navigation on a board, collect coins, drop bombs, do not kill itself, destroy the crates and play against other agents. The number of tasks that the agent must handle makes the game a complex decision-making problem. Scaling reinforcement learning to such complex sequential decision-making problems is a big challenge [2]. Solving such tasks with deep reinforcement learning brings new challenges. In other words, neural networks solve some problems and create new problems of their own. Some of them are discussed in the following Chapter.

5.6.1 Data distribution

In reinforcement learning, the data distribution changes as the algorithm learns new behaviours. That is problematic for deep learning methods, because they assume a fixed underlying distribution [2]. "The ground truth labels" are not fixed. Moreover, reinforcement learning works with sequences of highly correlated states, while most deep learning algorithms assume that samples are independent. Reinforcement learning also has a delay between actions and rewards. [2] solves these problems with a combination of Q-learning, a convolutional neural network and experience replay, which inspired us to use this combination in our approach. Experience replay addresses the problem of data inefficiency (see subsubsection 5.6.2). In combination with convolution networks, experience replay alleviates problems of correlated data and non-stationary distributions.

Convolutional networks also allowed us to exploit the translational symmetry of our game.

5.6.2 Data efficiency, learning efficiency

As already mentioned, our big difficulty was cost of computations. Reinforcement learning agents incrementally update their parameters and do not make use of experience, rapidly forgetting possible rare experiences and have strongly correlated updates that break [4]. Agents have to interact with their environment (which is very time-consuming) again instead of using more memory with rare experience (which is often cheaper). We tried to solve these difficulties by prioritized experience replay [4]. Experience is stored in a replay memory, that is why rare experience can be used in future updates and it becomes possible to break temporal correlations. It also saves computational costs, reducing the amount of experience to learn. Prioritizing which experience is better for the reinforcement learning agent to learn from, means that important transitions are replayed more frequently. This makes the experience replay more efficient, improves learning time and final policy quality, as compared to uniform experience replay [3].

The efficiency of learning the state-value function is one more difficulty. We used a dueling neural network architecture, which can more quickly identify the correct action during policy evaluation and to generalize across actions [3]. Such architecture can learn which states are valuable, without having to learn the effect of each action for each state [3]. This should be useful in states when our agent acts without affecting the environment in any relevant way (for example, navigating a board without gathering coins or dropping bombs).

6 Possible Improvements (Valentin)

Our approach to the problem roughly followed the combination of different strategies in [1]. We implemented deep Q learning with a target network, integrated a duelling neural network and used prioritized experience replay. Additionally, we used convolutional networks to exploit the translational symmetry of our game and further modified it to also exploit the rotational symmetry and the mirror symmetry of our game state. Lastly, we also added solutions to problems we encountered that were more or less specific to our game, e.g. avoiding static decisions, which can be applied to larger classes of games, and using some form of conditional exploration, which was implemented in a way that is harder to generalize to other problems.

The obvious way of improving our agent would now be to also incorporate the remaining improvements from [1]. The target network could be enhanced by using double Q-learning, i.e. by training it independently from the decision network, but this would also considerably increase training time. We tried to use noisy networks, since they would naturally apply the sort of conditional exploration we implemented manually, but ultimately did not have the time

to get them to work properly. Multi-step learning would probably improve the performance when using bombs, since this enables the agent to more easily learn complex sequences of actions. However, we did not have the time to implement this in a way that would not exceed the amounts of memory at our disposal. Distributional reinforcement learning would also provide a more natural way of exploitation.

Implementing a form of self-play would also have been useful, but we could not get it to work with the way we implemented our model. A better graphics card would have made that possible, or for example using Google Colab as proposed below.

In the end, we achieved an almost perfect performance on the first task and a sub-optimal one on the second task, while our final agent showed a performance that was far superior to that of the simple agent in the full game. Had we had the time, we would have trained a model for the first task again, since there was some room for improvement left, and continued training the model for the second task until it was comparable in performance to the simple agent. Also, while the final agent showed a formidable performance in the full game, it performed less optimal when evaluated on the prior sub-tasks. This could theoretically be overcome by training the agent further while reducing the learning rate, but we could also try to explicitly facilitate this by training our agent simultaneously on the different tasks. We tried this approach when training our agent from scratch, and the results were underwhelming. However, continuing the training of our final agent in this way could plausibly improve its performance on the sub-tasks, while keeping its superiority on the final task.

Perhaps most importantly, we neither had the time nor the computational resources to perform a more elaborate fine-tuning of our parameters than simply trying to guess them.

In general, our computational resources proved to be the strongest limiting factor, greatly limiting the amount of testing we could do. Since the state has to be encoded as a One Hot, and using convolutional networks does not allow us to exclude irrelevant degrees of freedom, i.e. walls that are not changing etc., training the network took a long time. Since we could exclude most of the state space for only collecting coins, training our model until reasonable convergence on this constricted space was possible within a relatively short time. However, with the full state vector, training our model on a mid-range GPU takes a few hours only until the ultimate effect of changes can be determined.

This is also the most important improvement we would suggest to the game setup. What we believe is the easiest way to access larger computing powers is to use Google Colab. Users with a Google account get free access to a Tesla K80 GPU and 12 GB of RAM for up to twelve hours at a time, easily enough to train a model of the size that could be required for our game. If the neural network can be modified accordingly, even a Google TPU can be used to train it. Being able to run the environment and the agents in this framework would have given us access to vastly more computing power than our own computers could provide. Sadly, we could not get our framework to work in this environment, which is essentially a jupyter notebook. The problem seemed to be pygame, so

we do not know how much work would be required to create a version of the game that could be used in this context. However, we think it would greatly improve this project if the training could be outsourced in this, or a similar, way.

Also, it could be useful to adjust the settings of the game from inside the running game. Adjusting for example the crate density or the amount of players from the main game loop can be implemented fairly easily, but providing this functionality could still help. We tried training our model on alternating games, i.e. with or without crates and enemies, for which we had to modify the environment. We ultimately abandoned this approach, but maybe other groups would like to try something like this. An example of a simple implementation can be found in our GitHub repository.

When evaluating our agent on sub-task one, we noticed that the game only ends when there are no more explosions ore bombs present. This is the right choice when playing the full game, since a bomb might still kill the remaining agent. However, when only evaluating the model on the first sub-task, this regularly inflated the number of steps our model needed to complete the task, since it still used bombs when they were useless. It would be useful to provide a Boolean in the settings for whether the game should end when there are still bombs and explosions present or not, especially since it is trivial to implement. Again, an example of a simple implementation can be found in our GitHub repository.

Lastly, the game state does not contain any information about the origin of specific bombs. Knowing whether it is the agent's or not would be useful information. This could theoretically be implemented from inside the agent, however we did not have the time to do so. It would be nice if the game could supply this information on its own, especially since it already keeps it to distribute points.

References

[1] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. G. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning", *CoRR*, vol. abs/1710.02298, 2017. arXiv: 1710.02298. [Online]. Available: http://arxiv.org/abs/1710.02298.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning", *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015, ISSN: 00280836. [Online]. Available: http://dx.doi.org/10.1038/nature14236.

[3] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning", 2016. [Online]. Available: https://arxiv.org/pdf/1511.06581.pdf.

[4] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay", *CoRR*, vol. abs/1511.05952, 2015. arXiv: 1511.05952. [Online]. Available: http://arxiv.org/abs/1511.05952.

[5] S. Dieleman, J. D. Fauw, and K. Kavukcuoglu, "Exploiting cyclic symmetry in convolutional neural networks", *CoRR*, vol. abs/1602.02660, 2016. arXiv: 1602.02660. [Online]. Available: http://arxiv.org/abs/1602.02660.

[6] R. Bellman, "The theory of dynamic programming", *Bull. Amer. Math. Soc.*, vol. 60, no. 6, pp. 503–515, Nov. 1954. [Online]. Available: https://projecteuclid.org:443/euclid.bams/1183519147.

[7] L.-J. Lin, "Reinforcement learning for robots using neural networks", 1993. [Online]. Available: https://apps.dtic.mil/dtic/tr/fulltext/u2/a261434.pdf.

[8] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization", *arXiv preprint arXiv:1412.6980*, 2014.

[9] M. Zhou, *Reinforcement learning methods and tutorials*, https://github.com/MorvanZhou/Reinforcement-learning-with-tensorflow, 2018.

[10] H. van Hasselt, "Double q-learning", 2010. [Online]. Available: https://papers.nips.cc/paper/3964-double-q-learning.pdf.

[11] H. Sahni. (2018). Reinforcement learning never worked, and 'deep' only helped a bit., [Online]. Available: https://himanshusahni.github.io/2018/02/23/reinforcement-learning-never-worked.html (visited on 02/23/2018).