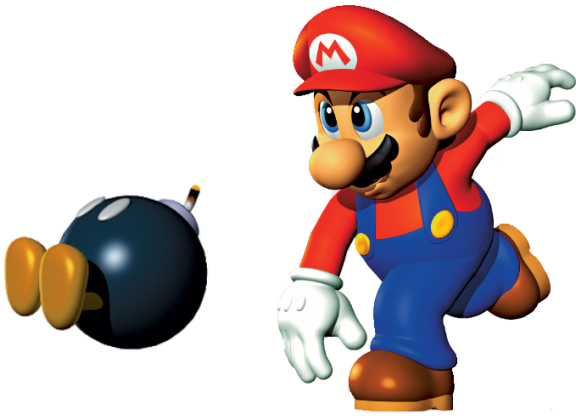
Super Mario Bomber

Richter, Daniel Habermann, Rhea Aßmann, Julian

April 4, 2022



Abstract

In this report we present our approach to develop a reinforcement learning model which successfully learned to control an agent in a given bomberman-game environment. Among the main concepts we explored were a Q-table agent as well as a Deep Q-Learning Network. The final task of navigating the environment and playing against multiple strong opponents was best performed by our DQN with an agent centred field of view incorporating several layers of information about the game.

Contents

1 Introduction

Rhea Habermann, Daniel Richter

In the lecture of "Fundamentals Of Machine Learning" we learned about the theory of Reinforcement Learning (RL) as another example besides Supervised and Unsupervised Learning and were then instructed to put it into practice. The given task was to develop two models that can learn to play (or in the ideal case, master) the game "Bomberman". In this report we want discuss the strategies used as well as the general development process of the two main models we used. However the structure in this report does not represent the actual timeline of events since we developed both models in parallel.

1.1 About Bomberman

Rhea Habermann, Daniel Richter

The game Bombermariooo can be described by a Marcov model as its future states (s_t) only depend on the current state (s) and is independent of the past. We further can observe all variables ($o_t = s_t$) and have, for the first two tasks, an almost deterministic policy ($a_t = \pi(O_t)$) as only the distribution of the coins and in task two the crates (with some of them containing coins) being random. By introducing awards (R_{t+1}) we get a full transition model ($p(s_{t+1} = s', R_{t+1} = r|s_t = s', a_t = a)$). Thus the fully observable game with a, for the first two tasks, almost deterministic policy can be described as a Markov decision process.
The game looses its deterministic policy in tasks three and four as other agents get introduced whose actions can not be calculated by our own agent since they act unilateral.
With these conclusions our Reinforcement Learning Problem can be described by an agent, which takes actions in an environment, which is perceived by sensors into a state and reward, which are then fed back into the agent.

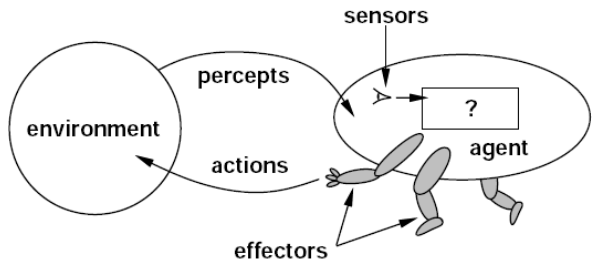


Figure 1: Surrounding Features[1]

Apart from these more theoretical traits (showing all qualities of a Markov Model) the game environment exhibits some more properties that will be used later on. One of the games characteristics are the symmetries regarding mirroring and rotation by 90° since the 17×17 field consists of different tiles that are ordered to create a regular grid.
The game either ends if the task (i.e. collecting all coins, destroying crates and defeating your opponents) is fulfilled or if the maximum of 400 steps is reached.

1.2 Evaluation

Rhea Habermann

In order to quickly observe the learning rate and status of our agents we used wrote a script "analysis.py". This script would print out the rewards our agent achieved (averaged over the last N played games, N varying in size depending on the current model used) and the episode length. We included some of the plots later in our report.
To see what exactly our agent has learned we used the **gui** or **log files** since the single information of episode length and rewards where fine to determine whether our agents directly blow itself up or not, but not enough to give us a good picture about our agents exact actions (e.g. whether it learned to just wait, run in loops or walk around seemingly random).

1.3 Further Libraries, Git Repository

We loaded our code to the git repository which you can find here:

https://github.com/Ztec1337/bombermario

Besides the in **main.py** already used libraries we used:

- **tensorflow**: used extensively for our DQN agent
- **pathfinding**: used to calculate some of the features needed for our Q-table agent
- **seaborn**: used to visualize data in a modern style

[1] https://www.doc.ic.ac.uk/project/examples/2005/163/g0516334/index.html

2 Getting started - A ta(b)le of Q's

Rhea Habermann, Daniel Richter

Given the information above we had to design our own agents which meant, we needed to pick a reinforcement learning method to go with. The lecture gave us a good introduction into Q learning which seemed to be realistically achievable which is why we started reading more about Q-tables and the ϵ greedy policy.
The Bellman equation was already known from the lecture defining the **value function** V_π of the current state s. The equation takes into account:

- the policy π for an action a in the state s
- the immediate reward r
- the probability for the following state s' and possible reward r at given state s and action a
- a discount factor γ ($\gamma \in [0, 1]$)

All in all the equation as defined in the lecture is:

$$V_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma V_\pi(s')\right]$$

We can include a factor α to modulate the learning rate ($\alpha \in (0, 1)$) used to update the action value function. The Q tensor is updated by combining the current Q-value, the reward (if an action a_t is taken in state s_t) and the maximum reward that can be expected in state s_{t+1}. To update Q we take its old value and add the rewards times the weighted maximum rewards subtracted by the old value, we weight the added temporal difference with the learning rate α. In summary the computationally used update function we implemented is given by:

$$\begin{aligned} Q^{(\tau+1)}_{(s_{\tau t}, a_{\tau t-1})} \leftarrow (1-\alpha)\ & Q^{(\tau)}_{(s_{\tau t}, a_{\tau t-1})} \\ & + \alpha \left(r + \gamma \max_a Q^{(\tau+1)}_{(s_{\tau t}, a_{\tau t-1})}\right) \end{aligned}$$

The factor γ is needed to adjust the future-rewards influence. We further included the epsilon greedy policy which meant we had to add a factor ϵ to the equation that determined the probability for our agent to make random steps during training. The probability as we used it would start at a high value and decrease during training.

2.1 First steps - Taks 1

Rhea Habermann, Daniel Richter

As proposed in the final project exercise sheet we started with a map cleared of all crates (the **coin-heaven** scenario sets crate density to zero and increases the number of coins) so we would only have 50 coins placed randomly over an empty map.
In this scenario we can ignore the "BOMB" action since we do not want our agent to blow it self up and hence only need to explore the actions "UP", "RIGHT", "DOWN", "LEFT" and "WAIT".
Since the Q-table reserves space for all possible state combinations, it is of most importance to reduce the possible feature space. With these conclusions in mind we first tried to implement the positional input by using the immediate 3×3 surroundings of our agent as a first set of features.

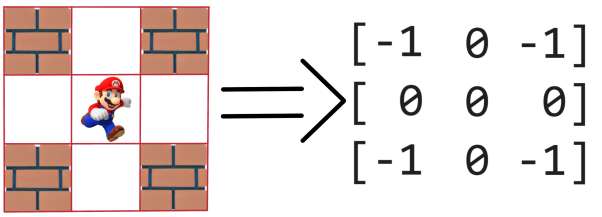


Figure 2: 3x3 Surrounding features

This however turned out to be very inefficient use of working memory space since most of the time some features imply the other features e.g. wall above $\iff$ wall below on most positions. To give our agent as little information as possible while at the same time providing all positional information needed to come up with a valid move we took advantage of symmetries. Instead of using the position with the entire surrounding directions we only differentiated into even and uneven positions, since those two properties hold most of the information about the map. That means, we divided our agents position by 2 and whether that was possible or not returned **1** or **0** for both fields above and below and similarly left and right respectively. For the 3×3 surrounding features the possible feature space can be calculated by:

$$N_s^{N_d} = 2^9 = 512$$

where N_s represents the number of possible states in each state and N_d the number of dimensions. With this new feature the number of total distinct states can be downsized to only $2^2 = 4$ which comes in handy later on. These positional features allow our agent to successfully navigate the game environment but not to collect coins. We furthermore wanted to stick to our idea of having an agent centred view and decided to go with a vector pointing towards the

next coin starting at our agents position. In order not to run into a problem while collecting the coin our vector just gives out (0,0) if the agent is at the field containing the coin, this only occurred in some starting positions. The total feature space with those two features combined is then given by

$$N_{\text{s-coins}}^{N_{\text{d-coins}}} \cdot N_{\text{s-pos}}^{N_{\text{d-pos}}} = 29^2 \cdot 2^2 = 3364$$

Since the next coin relative to itself could be located $\pm(14+1)$ tiles apart from himself.

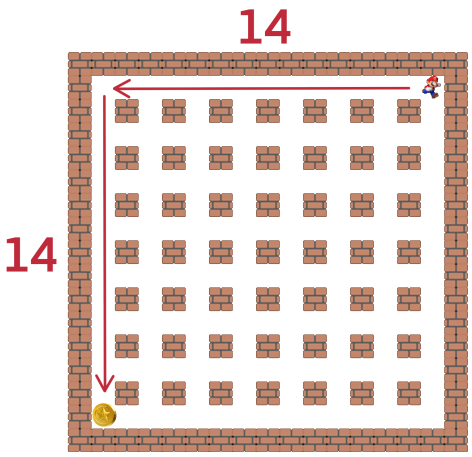


Figure 3: possible feature space for relative coins

In addition to setting the features we also needed to set our hyperparameters.
In order to try and avoid long waiting times we looked online for hints what good parameters may look like and tried orientating us at these values.
Set up with some ideas about the scales the parameters should follow, we still had to try out different values to see which did and which did not work for our game. (The other projects were about a bot, learning to walk or other mini-games but on the same level as Bomberman regarding the range of possible features.) This process took us quite a while but we hoped, that the fine-tuning we did now would help us with harder tasks in the future.
Since our model in the beginning has not learned anything and has no idea whatsoever of what would be a good or bad move, we thought that starting with a high ϵ would allow a lot of exploration and provide for a lot of different situations. However, while exploring a lot at the beginning is helpful, we want our agent to rely on the Q-table in later game states, thus our ϵ needed to decrease with time/rounds of training. We always started at a high value and would adjust the decrease according to how many rounds we wanted to train our agent.
But just to make sure our agent in training mode would never only relay on the Q-table and always explore some new steps we set a minimum of $\epsilon = 0.1$.

Our Epsilon Greedy policy is hence defined by:

$$\epsilon(t) = \begin{cases} \epsilon_{\text{start}} \cdot \epsilon_{\text{decay}}^t & \text{if } \epsilon_{\text{start}} \cdot \epsilon_{\text{decay}}^t \geq \epsilon_{\text{min}} \\ \epsilon_{\text{min}} & \text{if } \epsilon_{\text{start}} \cdot \epsilon_{\text{decay}}^t \leq \epsilon_{\text{min}} \end{cases}$$

where with each episode we gradually decrease the percentile of random actions. After some time varying the parameters and testing different combinations our final set of hyperparameters came to be:

$$\begin{aligned} \gamma &= 0.9 \\ \alpha &= 0.1 \\ \epsilon_{\text{start}} &= 1 \\ \epsilon_{\text{decay}} &= 0.999 \\ \epsilon_{\text{min}} &= 0.1 \end{aligned}$$

Besides the hyperparameters we also needed to set the rewards. This task definitely was easier since we felt like we had a better feeling for those parameters. So in this first task we only gave rewards for collecting coins and penalised making an invalid action.
With those features and parameters our Q-table successfully converged after about 2000 Episodes.

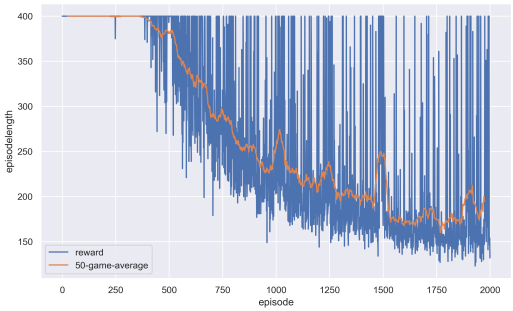


Figure 4: Episodelength over 2000 episodes

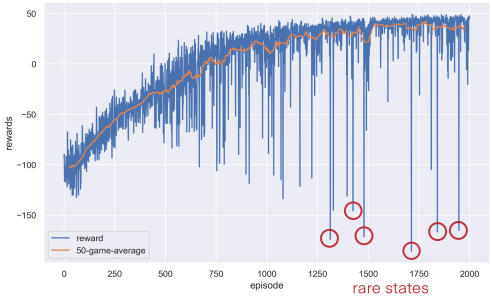


Figure 5: Rewards over 2000 episodes

Inspecting the logs of some games we observed our agent was able to collect a good amount of coins in the beginning of a game, but sometimes got stuck in later game states due to increased occurrences of exceptionally rare game states.
Since our trained model only went for the next coin it would move around the map a lot leaving behind

some coins, leading to rare game states which it has never seen before and being stuck. We tried to wrap our head around how to tell our agent not to walk off and instead clear one region after the other. We thought about including a coin density parameter for each region but since we needed discrete features our idea of a density-feature seemed too hard to implement so we neglected this thought.

In conclusion, the time needed to train our agent and first and foremost the storage used heavily depends on the number of entries in our Q-table. Minimizing them became an important task. With our selection we got 2 very discrete positional features as well as 2 good features for finding coins. Our first ideas for the features were pretty rudimentary. We tried giving our agent its surrounding and since it does not yet need to know how to escape a bomb and the coin density was pretty high we decided to go with the (un-)even feature and relative coin distance. We hoped it would give us a good start to work with while being relatively small.

2.2 Getting stuck - Starting Task 2

Rhea Habermann, Daniel Richter

In contrast to the **coin-heaven**- the **classic**- scenario did not provide the symmetries we used earlier on. Crates are spread randomly over the map with a crate density of 75% and in total 9 coins are hidden behind some crates. Another great change was the added **bomb ACTION** we now needed to use what evolved to be a major challenge.
Our first naive attempt was to just give our agent from task 1 the **bomb** action and reuse the surrounding tiles feature. The even and uneven tile feature would not work because crates are not spread symmetrically. This first attempt did not work as expected, so we went on and changed the squared input field into a cross and increased the field of view (**fov**) further. Since the next safe-space usually is not located in the immediate surrounding tiles we thought about different field of views. We considered a star shape, but since most of times this does not include lots of safe spaces, we thought about a 5×5, two tiles in each direction feature.

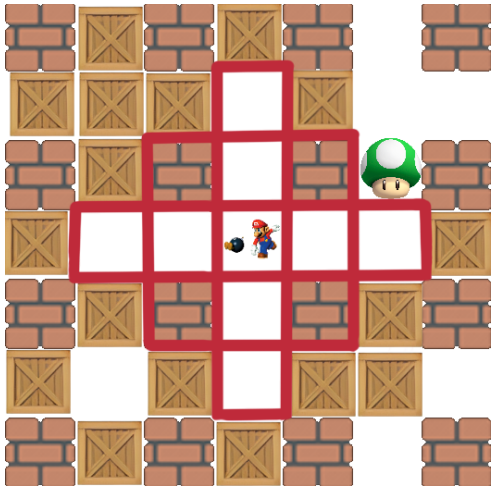


Figure 6: the problem with the **cross**-feature

The *1 up mushroom* here represents the save-space if the agent would now place a bomb. But the agent could not see it, in this scenario all directions seem to be equal.
However, combining these enormous features with our already large feature space for finding coins turned out to be slightly out of bounds for the given RAM constraints for the tournament. The required memory for the star shaped field of view Q-table can be calculated by

$$N_{\text{s-coins}}^{N_{\text{d-coins}}} \cdot N_{\text{s-fov}}^{N_{\text{d-fov}}} \cdot 4[\text{Bytes}] = 29^2 \cdot 3^{13} \cdot 4[\text{Bytes}] \\ = 5.363 \times 10^9[\text{Bytes}]$$

Where 4 Bytes is the required memory size for saving a single *int*, in our case, state for a given feature. For our state-fov the possible states are given by the feature map to be -1, 0 and 1 for walls, free tiles and crates respectively. And this does not even include any features regarding bombs or opponents. With these conclusions we realized our strategies for the Q-table had to change and we needed to define more discrete features.

2.3 Loosing a-way - Redefining Features

Rhea Habermann, Daniel Richter

To give our agent at least some orientation awareness information we reduced the length of the crosses arms and hence settled with a tiny cross length of 1. Instead of increasing the number of dimensions we increase the number of possible states per dimension and drastically decrease number of dimensions, resulting in 4 dimensions $x_i, i \in \{0, 1, 2, 3\}$ representing the directions “UP”, “RIGHT”, “DOWN” and

"LEFT" respectively.

$$x_i = \begin{cases} 0 & \text{if blocked} \\ 1 & \text{if Not blocked} \\ 2 & \text{if Not blocked and coin} \end{cases}$$

Where being **blocked** means the tile is occupied by either a wall, a crate, an opponent, a bomb or an active explosion. **Not blocked** meaning none of the above is the case and hence the tile can be walked on validly and not expect to die instantly. Lastly **Not blocked and coin** represents a very good state, where the agent should walk into that direction immediately.

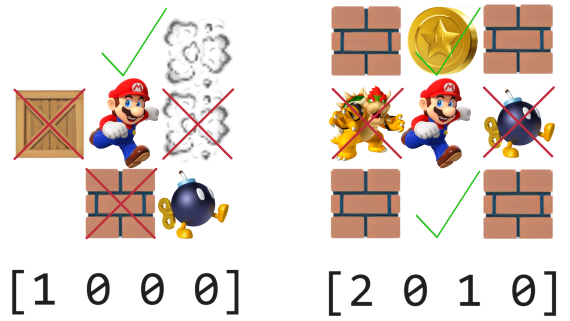


Figure 7: Surrounding Features

The most obvious and serious flaw was, that our agent was missing a bomb specific feature and thus, after placings a bomb, did not know when to place the next bomb or how to find save-spaces to escape to unless the safe-space is directly next to the agent. So after short periods of training the agent learned to not lay any bombs and instead stayed in small loops or waited in corners. Waiting in general first seemed to us as one of the biggest problems (that would change in the later development of the project) so we set the penalty for waiting even higher.
With our 4 features in mind we went on designing the bomb feature.

2.4 Finding multiple ways - Implementing the A* algorithm

Rhea Habermann

We figured it would be best to enlarge the number of dimensions rather then the number of possible states, since properties, like being able to drop a bomb, are largely independent of directions. Hence we implemented a new feature slot to represent bomb characteristics.
When looking in the list of important and therefore usable python libraries we found **pathfinding**. The pathfinding library has a function for the **A* algorithm** which we could utilize, rather than implementing one ourselves.
The A* algorithm combines the best of the **Dijkstra**- and **Breadth-First**-algorithms.

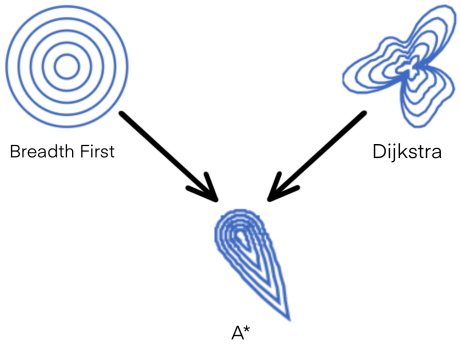


Figure 8: Draft of the algorithms[2]

The Dijkstra-algorithm looks at where the goal is and tries to connect the starting point with the end position by the seemingly shortest way. While this works fine in an environment without obstacles, the algorithm can´t find the shortest way if the linear distance is blocked. The algorithm can´t see the barrier until it has reached it and then needs to find a walk around which is seldom the shortest way but it is quite quick.
The Breadth-First-algorithm on the other hand would find the shortest way even with many obstacles. This algorithm starts its search by looking around it in continuously growing circles. It will expands its so called frontier until it has reached its aim but since blindly going in all directions without prioritising one direction, it is relatively slow.
Combining the best of both algorithms gets us to the A* algorithm. It prioritises the direction of the goal but still explores more of the map and thus finds (one of) the shortest way(s) possible. (There often are different ways that are equal in length.)
In detail it takes a function that estimates the distance to the next possible point (it is important, that the actual distance is either smaller or exactly the same as the estimation.) This process then reruns from all found points. Form there on it continuous, enlisting all explored fields in a table until it has reached the goal. In the last step, it makes it way backwards, so from aim to start, always choosing the shortest distance.
Here it is important to choose the **heuristic**, as the estimation-function is called, wisely. A heuristic that is close to or exactly the actual distance speeds up the algorithm but may not be easy to find for all cases. The most general heuristic would be the linear-distance as the distance can not be shorter than that.

[2] https://www.redblobgames.com/pathfinding/a-star/introduction.html

With that in mind we went on and used the **A*** algorithm for all pathfinding tasks.

2.5 Looking around - finding a safe-space

Rhea Habermann

We divided the task of finding a suitable safe-space up into two parts. For the first part we implemented a function which checks if a bomb in the current state would be a guaranteed selfkill and returned **True** or **False** respectively. For this task we simulated a hypothetical bomb placement in the current state and checked if there are any safe-spaces, that are tiles where no explosion from our own bomb will be present and are reachable from our current position.
Our first attempt to place a hypothetical bomb was to reuse the agent centered maps, introduced later on for the DQN, to hypothetically place a bomb in the center of a new explosion matrix and by changing all 0s (free tiles) and 1s (crate tiles) around our agent in a range of 3 tiles into a -1 (explosion) unless it meets a -1 (wall) step by step. We figured the most elegant way was to write it out once and then make use of the symmetries and repeat the steps after rotating by 90° for three times. With this new fifth feature the agent knows when to place bombs but still does not know where to go after a bomb has been placed. Furthermore this feature did not incorporate bombs placed by opponents.
While this worked fine for the bombs our own agent placed, the detail, that we used its position made it particularly difficult to copy the same mechanism for the other bombs. So we had to change our strategy and ended up introducing a new temporal field 6 steps ahead (in case of bomb placement: the four steps before the bomb explodes plus the two steps where the explosion is present), so the algorithm always knows where it´s save to walk.

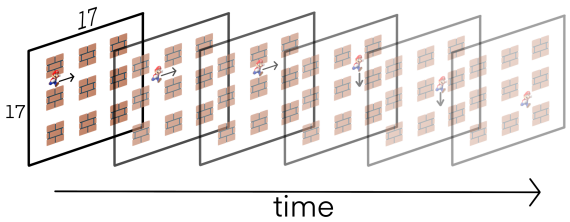


Figure 9: Forshadowing the next 6 steps

The general idea was to find a path so with all currently active bombs the agent would find a safe place to be 6 steps ahead. This dynamic time map would indicate save tiles to walk on and and would allow the agent to dodge all explosions.

Alternative to this solution, giving the model information about whether or not placing a bomb in this situation is a *safe-kill* or not could have worked as well. The model would have to learn to only place a bomb if the *save-kill-feature* is **False** and could then start following the **save-space-feature** in order to escape.
Equipped with these information the agent should able to play very strongly against opponents. It however still would need an exploring feature to find crates or hunt other players on the map.
This however became very complicated and since the approach seemed very algorithmic we stopped and rather went on with our DQN.

2.6 Going away - putting the Q-table aside

Rhea Habermann, Daniel Richter

We quickly ran into the problem of how far we wanted to go with the selected features. The game, as the *rule_based_agent* demonstrates, can be played well by a set of rules the agent has to follow. By implement those with very good preprocessed features it may not qualify as learning. With proper algorithms to preprocess the data, we are sure a Q-table would be fit to master the game and defeat most other agents. But it is a fine line to walk. To avoid this dilemma, we went on and further developed our second approach.

3 Deep Q Network

Rhea Habermann, Daniel Richter

3.1 An introduction into neural networks

Daniel Richter

Going with the Q-table did not work as planned. Towards the end it felt more like designing an algorithms, simulating future states and checking for the outcome. The project asked for a second model and as we were not sure how far we were allowed to go with algorithmic calculated features we went on and started to think about what other model we can pick for our second agent. Neural Networks sound fancy so we went with **Deep Q Learning**.
An **Artificial Neural Network**, from here on referred to as **NN**, is a network of nodes with weights inspired by the way neurons in animal (or human) brains work.

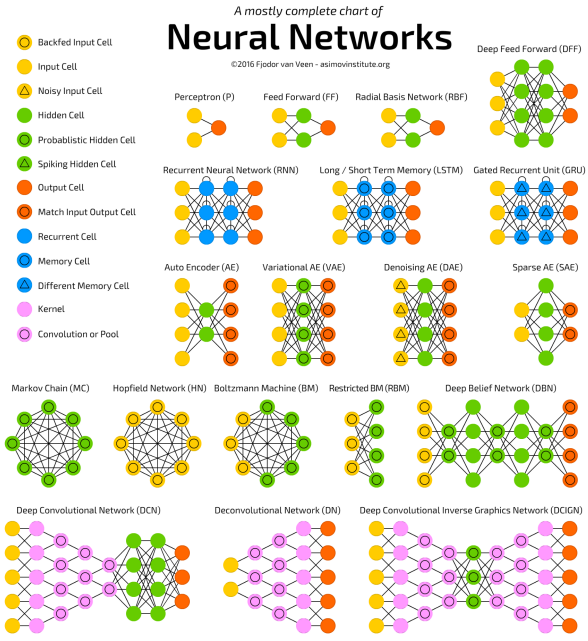


Figure 10: A sample of NN[3]

There are plenty of NN's to choose from, but to understand the basic concepts we first choose two simple hidden dense layers and very small trainable variables spaces for task 1. After getting more of a feeling for NN's we then switched to a more complicated for the latter problem more suitable **Convolutional Neural Network** structure.
CNN's are commonly used in image analysis with the image being the input. Since we use an agent centered field of view and similarly to RGB-images make use of different input channels, this seemed like a comparable case of use.

3.1.1 DQN-Theory

Daniel Richter

For our purposes, we adapted a Deep Q-Network algorithm which was first developed by DeepMind in 2015. The algorithm enhances Q-Learning to incorporate deep neural networks and a system called experience replay or replay memory.

Deep Q-Learning

Correlative with our earlier approach using Q-tables, the basic idea of Q-Learning is to use Bellman's equation to iteratively approximate an optimal Q-function.
For increasingly large feature spaces, it becomes impossible to store the Q-function as a table containing all state and action combinations. With Deep Q-Learning we try to use a NN as a function approximator, to estimate the Q-values for each action.

$$Q(s,a)^* \approx Q_\theta(s,a)$$

where Θ represent the parameter of the NN.
This Q-network can be trained by minimizing the expected loss functions

$$L_i(\theta_i) = \mathbb{E}_{(s,a,s',r)\sim\rho}\Big[(y_i - Q_\theta(s,a))^2\Big]$$

where $y_i = r + \gamma \max_{a'} Q_{\theta_{i-1}}(s',a')$ is the so called temporal difference target and ρ the behavior distribution. Differentiating the loss with respect to the weights, we can calculate the gradient, which we apply using the autodifferentiator GradientTape[4] as defined and provided by tensorflow.

$$\nabla_{\theta i} L_i(\theta_i) = \mathbb{E}_{(s,a,s',r)\sim\rho}\Big[\Big(r + \gamma \max_{a'} Q_{\theta i-1}(s',a') - Q_{\theta i}(s,a)\Big)\nabla_{\theta i} Q_{\theta i}(s,a)\Big]$$

Here it's computationally advantageous to use stochastic gradient decent instead of computing the entire gradient.
For reasons regarding stability we do not alter the NN parameters θ_{i-1} in every step. Instead we update this so called *target network* a magnitude of iterations slower than the respective *action network*.

Replay memory

It is benifical to introduce a replay buffer. In this memory we save a large sample of transitions consisting of (s,a,s',done) the state, action, resulting state, reward and a *bool* indicating if it's the last

[3] https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464
[4] https://www.tensorflow.org/api_docs/python/tf/GradientTape

game state or not. For training we can then use a random batch sample of our memory. Since random samples are uncorrelated, this increases stability and by using transitions multiple times, we can increase data efficiency. In summary this results in the following base structure;

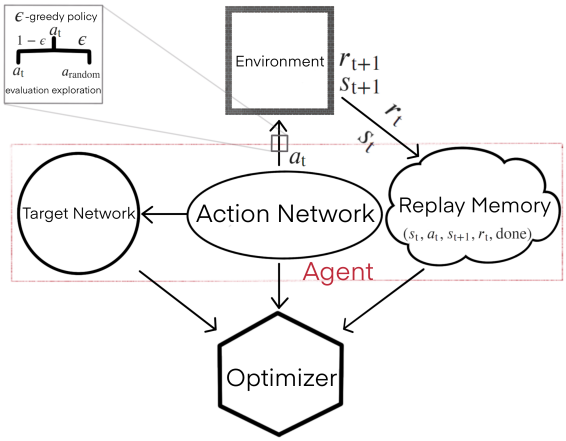


Figure 11: Deep Q-Learning structure

To introduce exploration, we add the ϵ -greedy policy, constituting Deep-Q-Learning, like Q-Learning as an off-policy algorithm.

3.1.2 Going towards Learning - Back propagation explained

Rhea Habermann

We want to shortly talk about back propagation as this is the most central part in NNs.
We assume training data represented by the tupel (x, y) is put into the network generating an output $f(x, w)$ where w stores the parameters of NN. The loss-function $l(.,.)$ takes the true class t and output of the NN.

$$l(t, f(x, w))$$

The loss function looks at every parameter individually. If we were to minimize it for each parameter the found parameters would only work well for one input or the different loss-functions would overwrite each other resulting in random outputs. Instead, the sum over the weights of all loss-functions needs to be minimized.

$$\underset{w}{\operatorname{argmin}} \sum l(t, f(x, w))$$

This can be achieved by gradient descent. In the ideal scenario we find the global minimum of the loss-function with this iterative first-order algorithm.
To apply this algorithm the function must be differentiable and convex. For our case we will assume that these properties are given. The new weights w^{new} depend on the old weight w^{old} and can be calculated by:

$$w^{new} = w^{old} - \eta \frac{\delta l}{\delta w}$$

The parameter η controls the step size and therefore needs to be chosen with great carefulness if not any local, but the global minimum should be found within a reasonable amount of time. The first case occurs if η is too big while a too small η extends the time needed to calculate the output. With every function being different, this parameter needs to be selected with respect to the individual function.
To demonstrate, that the deviation of the function is independent of the weight we take an example.

$$l(t, \sigma(w_2 r(w_1 x))$$

x still is the input, r represents a rectifier and σ the activation function.
Comparing the derivations of l by w_1 and w_2 respectively after applying the chain rule leads us to:

$$\frac{\delta l(t, \sigma(w_2 r(w_1 x)))}{\delta w_2} = \frac{\delta l(t, \sigma(w_2 r(w_1 x)))}{\delta \sigma(w_2 r(w_1 x))} \cdot \frac{\delta \sigma(w_2 r(w_1 x))}{\delta w_2 r(w_1 x)} \cdot \frac{\delta w_2 r(w_1 x)}{\delta w_2}$$

$$\frac{\delta l(t, \sigma(w_2 r(w_1 x)))}{\delta w_1} = \frac{\delta l(t, \sigma(w_2 r(w_1 x)))}{\delta \sigma(w_2 r(w_1 x))} \cdot \frac{\delta \sigma(w_2 r(w_1 x))}{\delta w_2 r(w_1 x)} \cdot \frac{\delta w_2 r(w_1 x)}{\delta w_2} \cdot \frac{\delta w_2 r(w_1 x)}{\delta r(w_1 x)} \cdot \frac{\delta r(w_1 x)}{\delta w_1} \cdot \frac{\delta w_1}{\delta x}$$

It becomes clear, that the first part of the deviation by w_1 is identical to the deviation by w_2. We can expand this correlation to further weights and recycle the terms from earlier results.

3.1.3 Starting all over again - CNN explained

Rhea Habermann

At this point we had studied the lecture and understood how the Q-table works, did some research about different Q-table strategies and feature development options. This whole process now needed to be redone.
Neural Networks (or NNs) as opposed to Q-tables are capable to work with a large input such as images or in our case the whole game board represented by a matrix encoding information about walls, coins, crates, bombs and players.
A classical input of a CNN is a 2d or 3d matrix that is fed through a convolutional layer or a series

of convolutional layers. In theory one could include pooling layers to scale down the amount of data but this becomes more important for larger inputs. Our input is already small enough for the network to work with so we did not implement a pooling layer. Besides, the pooling layer, or at least the max-pooling, in our case would cause an enormous loss of crucial information. In e.g. Max pooling the maximum of the considered region is taken, this makes since in image analysis (where CNN are often used) but in our case would make for all walls (represented by -1) to disappear because no matter where on the field, there is always a free tile (0) or a tile with a coin(1). To effectively navigate our agent needs the exact position of all tiles which is why we can't use pooling layers. Instead we only use convolutional layers. The convolutional in CNN means, that the data is processed by a kernel sliding over the input in essence resulting in a convolution as the output of one convolutional layer. Thereby the number of weights in the network is much smaller than in a fully connected network.

3.1.4 Implemented activation functions

Daniel Richter

In order to determine the output of a single node, a transfer function usually called activation function is needed. There is a variety of commonly used functions, however most of which can be categorized into linear and nonlinear activation functions.
All of in our network, or at least at some point, utilized activation functions were:

- **reLu** : rectified Linear Unit
  One of the, if not the, most widespread activation functions, commonly used in CNN's for image classification.

$$f(x) = \max(0, x)$$

- **linear** : linear identity
  The identity function.

$$f(x) = x$$

- **tanh** : hyperbolic tangent
  Can be useful to limit output to $(-1, 1)$

$$\tanh = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- **softmax**
  The softmax function can be interpreted as probabilities instead of the otherwise fixed values.

$$\text{softmax(x)} = \frac{e^{x_i}}{\sum_{i'} e^{x_{i'}}}.$$

In our case the softmax is used to replace the ϵ-greedy policy by choosing actions not completely random, but selecting actions according assigned probabilities.

3.2 Starting to walk - Task 1

Daniel Richter

3.2.1 Introduction

Daniel Richter

Since NNs are fairly complicated it seemed logical to start with the easiest **coin-heaven** scenario. At that point we already had a working Q-table and decent features. Considering the long time NNs need to train, we, instead of letting the NN learn powerful features on it's own simply reused the already implemented features used by our Q-table in task 1. We knew this had to work and hoped it would not take too long to train, since these relative coin direction and even/uneven positional features already worked well prior. To get familiar with general NN structure and still kept it simple, we chose two hidden layers with 8 nodes:

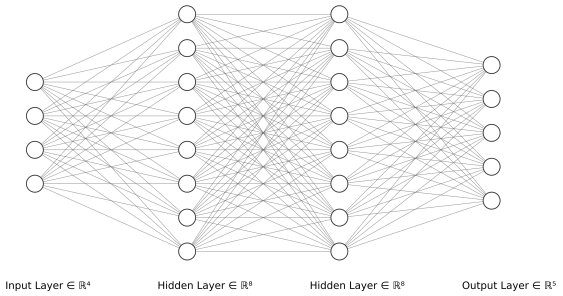


Figure 12: Our first NN[5]

After each but the last hidden layer we used a *"reLu"* and for the last layer a *"linear"* activation function as given by standard literature examples from https://www.tensorflow.org/. After short testing this structure and features converged, but there was still lots of potential for improvement by further developing features, and shaping rewards and hyperparameters. A feature we thought about but were not sure to work in a Q-table was, to normalize the nearest coin vector. Since states in Q-tables have to be discrete, we figured this would not work since the feature data format would change from *int* to *float*. Later we realized it might have worked anyways since there is only a finite number of normalized relative

[5]huge thanks to alex for providing easy framework to visualize NN's http://alexlenail.me/NN-SVG/index.html

nearest coin vectors inferred by the size of our map. However a NN based on a completely different methodology of learning, in simple terms does not care about discrete inputs and hence would be able to work anyways with this improved feature vector. By using a normalized relative coin vector many equivalent states can be represented by only a single state and therefore significantly reduce the amount of possible states.

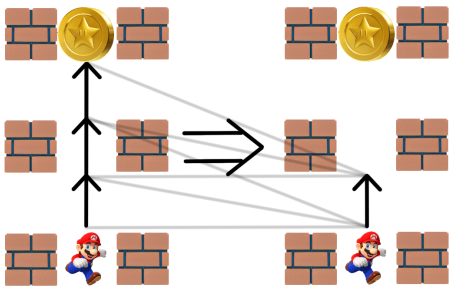


Figure 13: Squeezing the feature vector

Having established a base structure and precalculated features, the next step was to specify hyperparamters and shape rewards.

3.2.2 Optimizing Hyperparameters

Daniel Richter

Common literature regarding RL suggest to start training with 100% random exploration and implement decay over time. In this case we recycled the already written decay process from the Q-table before.
Since we only use part of Bellman's equation, only the discount-factor γ needs to be set. A common range of values across different literature examples were $\gamma \in \{0.9, 0.99, 0.999\}$. Our environment and feature space seemed to be quite small in comparison to other examples, and with a limitation of 400 future steps at most, $\gamma = 0.9$ seemed to be reasonable.
For batch-size the standard value as defined by tensorflow is 32. This seemed reasonable for our case as well, but to speed up training we varied the size later on. Our final set of parameters was hence chosen to be:

Parameter	Value
γ	0.9
ϵ_{start}	1
ϵ_{decay}	0.99
ϵ_{min}	0.1
learning_rate	0.001
batch_size	32
replay_memory	1000
update_action_network	5
update_target_network	500
optimizer_function	adam
loss_function	mse

Table 1: Hyperparamteters

where some were optimized by training the model and checking the learning progress, e.g. if the model learns at all, and others either influenced by literature or being the default value for a given parameter. For rewards we used the same rewards as previously defined for our Q-table.

Event	Value
COIN_COLLECTED	1
WAITED	-0.1
INVALID_ACTION	-1

Table 2: Events and Rewards

3.2.3 Evalutating our first steps

Daniel Richter

With those final parameters set, we can inspect and compare our training process with that of the Q-table. During training the main metrics we tracked were rewards and episode lengths, which provide reasonable insights into model performance.

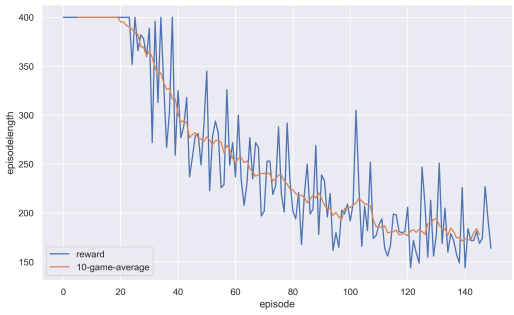


Figure 14: Episode length over 150 episodes

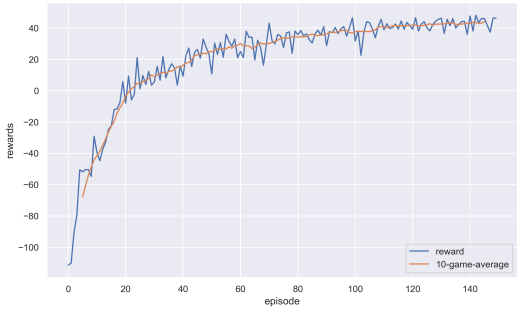


Figure 15: Rewards over 150 episodes

The results are truly astonishing. With optimized parameters the DQN model outperforms the preceding Q-table by a magnitude of metrics. Not only does it learn significantly faster in terms of episodes needed, but the learning process is much more stable as well.
There are several reason for the noticeable improvement. Particularly the improved normalized feature vector greatly reduces the number of states making it easier for the model to adapt much less frequent rare states. Furthermore the model get updated every 5 steps, but uses a batch size of 32 randomly selected transitions from memory. This means a single transition will be used multiple times ($\sim 6\times$) to update the model instead of only once. For the Q-table this would not help very much, but for our NN it does make a notable difference.

3.3 The final network

Rhea Habermann, Daniel Richter

After successfully solving the first task with both a Q-table as well as a DQN we started to look into the following tasks.

3.3.1 A failed attempt

Daniel Richter

With these precomputed well defined features a few simple dense layers may have been sufficient for completing the coin-heaven scenario. However, for a significantly harder task like blowing up crates, dodging explosions and collecting coins, a more sophisticated NN would be needed. Our first intuitive approach consisted of reducing the map-size to 11×11 feeding the entire game, consisting of a crate and wall layer, an explosion map layer as well as a player bomb layer where 1 represents us as a player and -1 represents our bomb, into a CNN. This approach seemed reasonable since we can interpret the game as an image with different channels representing different properties.
This model worked surprisingly well, especially in early game states, but failed to adapt to later game-states.
Considering the model already struggled to learn endgames on a small game map, we changed our strategy. A fundamental flaw with this approach comes from the metaphysical assumption where our agent was located on the map. We as humans may understand the correlation between a 1 in a matrix and a certain position on the field, but for a NN this might be a lot harder to learn.
We figured an agent centred field of view would solve this issue, though we would have to rewrite substantial parts of our NN input.

3.3.2 Which features we chose

Rhea Habermann, Daniel Richter

Having learned a lot about NNs up to this point we started to think about architectures and training methods to apply for the latter tasks. Although we trained multiple models on multiple computers, utilizing tensorflow's-CUDnn CPU-computing compatibility, training took quite some time. Since it did not make much of a difference to implement opponents into the input layer, we from here on started to train our models against multiple opponents, usually a rule_based-, a peaceful- and a random-agent.
For input we thought about various different fov-sizes, but after some testing we settled with an input of 9×9 in an agent centered view. Bombs have an explosion range of 3, so 4 tiles in each direction enable the agent in theory to *see* all relevant bombs and learn to dodge them. For corners and edges we expanded the map and filled all out of bounds tiles with zeros.

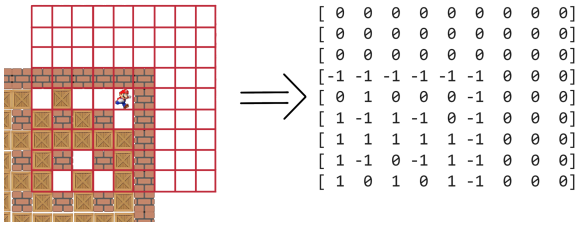


Figure 16: Expanded feature matrix

In summary this approach resulted in a $9 \times 9 \times 4$ feature tensor, of which each dimension represents some important property of the game:

- *1st dimension*
  The already utilized field map of the game dictionary.

$$d^{(1)}_{x,y} = \begin{cases} -1 & \text{if wall} \\ 0 & \text{if free} \\ 1 & \text{if crate} \end{cases}$$

- *2nd dimension*
  The explosion map given by the current game state.

$$d^{(2)}_{x,y} = \begin{cases} 0 & \text{if no explosion} \\ -1 & \text{if explosion} \end{cases}$$

- *3rd dimension*
  A map representing coins and bombs. Each position of coin in the fov will be set to a 1, while bombs are represented by:

$$d^{(3)}_{x,y} = \begin{cases} 1 & \text{if coin} \\ 0 & \text{if no bomb} \\ \frac{-1}{(1+b_t)} & \text{if bomb} \end{cases}$$

  where b_t represents the bomb timer.
  The idea was to represent potentially dangerous areas with increasingly negative values, since simply putting a bomb timer proportional value would result in a more complicated scenario. In the latter case, a high value would indicate less danger, a low value great danger, but 0 no danger.

- *4th dimension*
  A map for locating opponents and learning when to place a bomb - a bomb switch.

$$d^{(4)}_{x,y} = \begin{cases} 0 & \text{if no player} \\ -1 & \text{if opponent} \\ 1 & \text{if placing bomb is possible} \end{cases}$$

  where the bomb switch only affects the center tile, since no opponent can be located there.

The last bomb switch feature was added during training while we realized the agent did not know when to place a bomb and would spam 3 invalid actions in a row, trying to place a bomb.
This 9×9 agent centered field of view across multiple channels establishes a decent input for our NN. While it may seem like a general sense of position on the entire map seems to be missing, this resulted in some interesting behaviors, discussed later on. At this point we simply hoped, a 9×9 fov would be sufficient to *see* far enough to hunt all crates and opponents.

3.3.3 Network architecture

Daniel Richter

With our input layer set, we can implement the next layers of the network. We tried different layer sizes, kernel sizes and activation functions, but eventually settled with a structure similar to to the DCN as seen in figure 10. Three hidden Conv2D[6] layers with reLu activations, followed by two dense layers with one reLu and one final linear activation function.

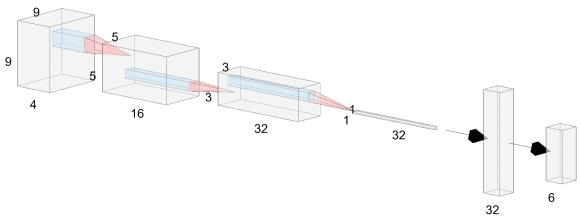


Figure 17: Final network architecture

One unfortunate problem we ran into multiple times came from apparent overfitting while implementing a softmax. Using a final softmax activation layer, commonly used for image classification and generating probability distributions, the model, at least in some occasions was excessively overconfident to take a certain action. This sometimes resulted in probability distributions where one action had a probability far beyond 99.98...%, essentially acting like an argmax function. Because of this the agent stopped exploring and was not able to learn. We tested several ideas to (?) fix this problem. Among the most promising attempts were:

- Reducing the NN size can generally help to reduce overfitting.
- Using softmax externally onto the output produced by our NN.
- Changing reLu activation functions of preceding layers to tanh activation, and limiting the prior output to (-1,1). This could have helped

However, none of the above worked as hoped so we stuck to the best working model and tried to improve other areas.

3.3.4 Improving training

Rhea Habermann

Running - exploring the final map

In the first try, we had seen, how task 3 and 4 are very similar to task 2, this is why we now merged these tasks and not only added crates but directly included enemies as well.
The rewards we used are the following:

[6] https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D

Event	Value
COIN_COLLECTED	5
KILLED_OPPONENT	10
INVALID_ACTION	-1
WAITED	-0.4
MOVED_DOWN	-0.1
MOVED_LEFT	-0.1
MOVED_RIGHT	-0.1
MOVED_UP	-0.1
BOMB_DROPPED	-0.3
OPPONENT_ELIMINATED	4
CRATE_DESTROYED	2
COIN_FOUND	2
KILLED_SELF	-2
GOT_KILLED	-3
SURVIVED_ROUND	1
LOOP	-1
KILLED_BY_OPPNENT	-5

Table 3: Events and Rewards

Following a role model (Luigi)

When starting the training process, the model learns by trying out different steps and therefore we need it to make random steps. However, especially in early stages of the game there are only few valid steps and even less good steps. The network takes a lot of time to learn only the basics because the agent in most cases blows itself up rather then making enough good steps in a row to stay alive.
This is unless the agent gets at least some information about what a good step in this situation may look like. Again, it is important not to give too much information as the agent could learn the rules instead of coming up with its own strategy. One of course would need a role model that knew how to play the game, here the rule_based_agent came in handy.
From now on some random actions weren't completely random anymore but suggestions of the rule_based_agent. The desired effect occurred and the model gained higher rewards in a shorter time period. The ϵ-decay in comparison is shown below.

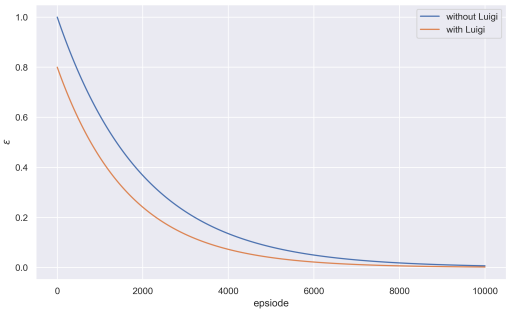


Figure 18: ϵ-decay

Becoming more consistent - optimizing the replay memory size

Our model at current state was quite good but did vary a lot what bothered us. When looking for reasons we stumbled upon the experience memory size that was set to 100. It seemed reasonable to enlarge it in order to fix our problem since we guessed that looking at more events rather then the 100 most resent episodes. Not wanting to exaggerate we decupled the experience memory size. After running the model again, it seemed, it has stabilized. Hoping this was a tendency we repeated the augmentation and went from 1000 to 10.000. The tendency remained and we went on up to 100.000. Now, the tendency got lost and we settled with a experience memory size of 10.000 as this remained as the maximum.

Going astray: catastrophic forgetting

One thing we did not expect as it is not common amongst humans or animals is a phenomenon called **catastrophic forgetting**.
Not knowing what we would get ourselves into we trained our model, while monitoring its rewards we were quite happy to see them grow continuously but this happiness was of short time because the NN started to worsen and would not stop getting worse and worse. Unprepared for this kind of conduct we did some research and discovered that at least we are not alone with this problem. Apparently NN can get worse if the states they are training in are all very good, it can predict its next steps confidently and gets high rewards. Since the network overwrites its memory this good sequence of events become all that it knows. Our model simply forgot what bad states look like but still came across bad states were it did no longer knew what to do but still predicting its next step confidently. The NN now learned, that its predictions were false resulting in a high error.
We can not say what exactly happened but this sounds like a good explanation for an otherwise weird circumstance. What makes it even harder to understand is that additionally we do not know which states the NN has linked together and therefore can not counter them. In some cases the NN relearns the good states but in other cases it doesn´t and has to start from scratch. We decided that either way it had to be trained and wanted to rather restart with slightly changed parameters.

Running away - training against too strong agents

When letting the agent train against three rule_based_agents we encountered a minor inconve-

nience with an unforeseeable outcome. The plan, to teach the agent more strategic behavior while playing against stronger opponents backfired. Our pretrained model was strong enough to survive up until most crates were destroyed and most coins collected but significantly weaker then the three rule_base_agents it had to compete. This resulted in a behavior we called **suicide-feature**. Our agent would first flee as soon as an opponent came closer and when trapped in a corner kill itself rather then getting killed.
While this might be helpful when playing against stronger players we did not encourage this behavior since we do not expect the majority of the other models submitted in the tournament to be as strong or stronger than the rule_based_agent.
This incident showed us, that we also needed to add our opponents strength into the equation.
One thing we realized to late was that this effect could also be used to teach a pretrained model certain behavior.
Due to time-issues we did not carry out the idea but think that reducing the crate and coin density to simulate the late-game could have taught the agent some strategics and made it a more efficient killer. One scenario we did develop and train shortly but did not work as hoped, is a deathmatch game mode. In it there are no coins or crates and the players play against each other. After short times of training the agent got better at dodging bombs but lost his ability to blow up crates.

Checking multiple ways - training four agents at a time

Training the network takes up a huge amount of time and waiting for several hours just to get some information without any comparable data is not quite constructive. Running everything again meant waiting once more just to then make slight changes in the hyperparameters. This attempt obviously is not specifically satisfying and speeding up this process thus became more and more important to us as the amount of rounds needed to see whether or not the model converged became larger and larger.
Just as mother nature does, we decided to run the same model with slightly different hyperparameters. This way we would have to wait only once and directly get four comparable outputs. Before training for another time we could change the hyperparameters to the once of the best performing model.
We called this process to fine tune the hyperparameters **evolutional hyperparameter fine-tuning**.

Looking around - different behavior depending on the agents position

At some point during evaluating training sessions we had to observe our agent playing astonishingly well when starting in upper corners while blowing itself up when starting in lower corners. At other times the outcome was the same with switched positions.
This still is weird but possible reasons could be problems with training sets, maybe it is not as random after all and some early stages directed the learning process into a local minimum.
A further reason we thought of could come from the non-convexity of our loss landscape. The network may converge to different local minima depending on initialization, early training samples or other influences.
Although not quite understanding as to why this happened we found a work-around to this problem.
Instead of giving the playing agent only the actual scenario we could see on screen, we let it compare the instructions it would get for the rotated scenarios before deciding what to do.

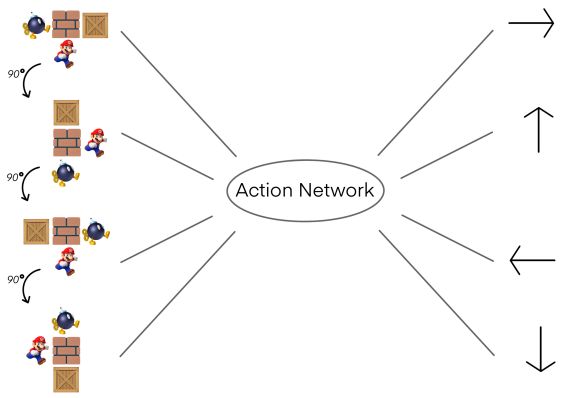


Figure 19: Rotation by 90°, 180°, 270°

This meant we first needed to rotate the field, let the model decide what to do and derotate the output. This was done by permutation π of the states.

$$\pi = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 3 & 0 & 1 & 2 \end{pmatrix}$$

We used this permutation for all 3 rotational operations by applying π 1, 2 or 3 times respectively.
This idea nearly solved the problem. Only in some cases did this lead to a lot of invalid actions, we made out the reason to be one overconfident state that would overshadow the other outputs. This flaw was rectified pretty quickly as well by normalizing the outputs before comparison.

Changing the paths - combining different models

During our training we followed mainly one path, meaning we trained a model, looked at what it had learned and tried to improve it by changing some of the training parameters.
This, as mentioned above, takes up a lot of time. In later stages we were fond of our remaining time so we started thinking about alternatives. We for example noticed our agent leaving behind a lot of coins but collecting them is a crucial part of the game. Remembering task 1 and the corresponding model that was trained specifically to collect coins we hoped by combining these two models would quickly teach our agent to prioritize coin-collecting. In fear the agent would get blown up if the coin-collecting model was to strong we weighted it with 20% for start. As this was still too much we gradually decreased it until we reached 1%. As to when this extra opinion was asked we guessed that switching from only task4 agent to added task1 agent every time there was a visible coin would destabilize the agent so the coin-collecting was always active. Since we used the features of task1, only walls are accounted for, making the agent effectively blind to crates. Even with the small weight of only 1% the extra model would often tell the agent to do invalid actions or capture it in a loop.
As those problems arose, we neglected this idea since we did not have too much time left and would have to reuse the already trained models that have the same structure as our final model.
However we kept thinking about the general idea and are of the opinion the fundamental approach might work. In a scenario where one has the time to train multiple models for different scenarios respectively. Where one model is especially good at destroying crates, one is trained only to collect coins and a third one stepping in to kill opponents. This could be achieved by altering the field as explained earlier in the report.
According to the current status of the game during play-mode the agent could take one of these three paths. As we did not have the time to follow this idea we unfortunately can not state whether or not this would have worked but nevertheless the idea seemed very promising.

Making better steps - adding new events and avoiding loops

Our agent often got stuck, running in a loop. These loops often prevented the agent from clearing the whole filed of all crates and collecting all coins because it would just run up and down or left and right, never leaving these two tiles for the rest of the game. Ultimately we created an extra event **loop** so we could penalize it for making the same three steps in a row.
As predicted, the agent stopped running in these small loops, it instead found larger loops it could run. That is not an exceptional improvement but it is one.
At one point during the process our model actually run unusually large loops, where it placed a bomb, went forward and repeated this process thereby walking across the whole field. We called that the **exploring-feature**, this seemed to be the best-case scenario. Sadly this model was not the most effective with regards to dodging bombs, collecting coins or killing enemies so we did not use it, of course hoping for the best model to learn the same but since we did not know how and why it developed this behaviour we could not force the other model to learn the same and had to settle with only the second-best case.
We, too late had an additional idea to discourage loops to actually try it out since we can not predict the behaviour the model would learn. Running out of time we decided not to give it a try but here is the general idea:
If stuck in a loop the agent visits the same field multiple times and fixing smaller loops is not a problem but as soon as the loops get more complicated then going back and forth we can not define different events for all possible loops. Instead we could store all tiles the agent had visited and penalize visiting the same tile more than three times with increasing penalty each time it visits the same spot. The penalizing would only start after three times since the agent sometimes needs to get back to where it came from to dodge a bomb.
Discussing the problems than come with this method we understand that its outcome is not trivial. The model could unlearn following one track or become worse at dodging bombs or way different stuff. Maybe dropping the earlier visited fields (a loop that is longer than ≥ 50 steps e.g. would not be recognizable for us) or setting the threshold for how often a field may be visited higher would be better proposals.
Or the opposite may work, giving awards for every field that has not yet been visited. We do not know what would happen but it would be interesting to see the effects.

3.4 Evaluation

Rhea Habermann, Daniel Richter

A sample of promising checkpoints we further inspected, by playing against 3 rule_based_agents, performances were:

	coins	totalreward	kills
base_model	169	234	13
ckpt-1	210	265	11
ckpt-2	167	207	8
ckpt-3	173	208	7
ckpt-4	129	174	9
ckpt-5	96	146	10
ckpt-6	140	190	10
ckpt-7	95	120	5

Table 4: Comparison of the best Checkpoints

The agents played for 100 rounds against their opponents.
Our final fine-tuned set of hyperparameters after many iterations hence came to be:

Parameter	Value
γ	0.99
ϵ_{start}	1
ϵ_{decay}	0.9995
ϵ_{min}	0.1
imitator_start	0.8
imitator_decay	0.9999
learning_rate	0.0001
batch_size	16
replay_memory	10000
update_action_network	4
update_target_network	2000
optimizer_function	adam
loss_function	huber

Table 5: Hyperparameters

We tried different loss_functions but consulting literature example, we stuck with a Huber loss function, since it is less frail to outliers and commonly used in Deep Q-Learning examples.
The rewards we defined can be seen in Table 3 *Events and Rewards*.
Using evolutionary-parameter we did not document trans-generational analytical metrics. We only recorded the model by taking checkpoints every 200 steps. For our latest training model one can nevertheless see some convergence.

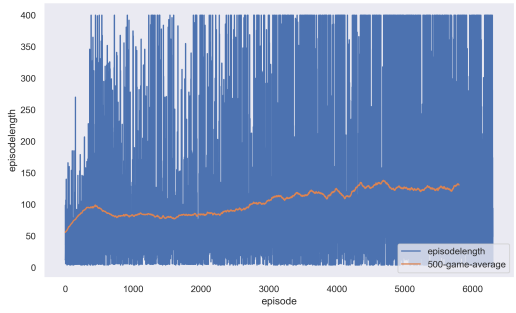


Figure 20: Episode length over 6000 episodes

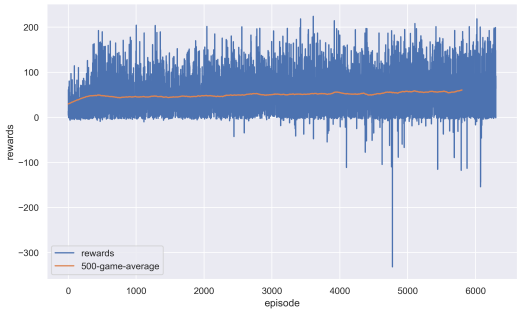


Figure 21: Rewards over 6000 episodes

The final structure used to train four Deep Q-Networks in one environment:

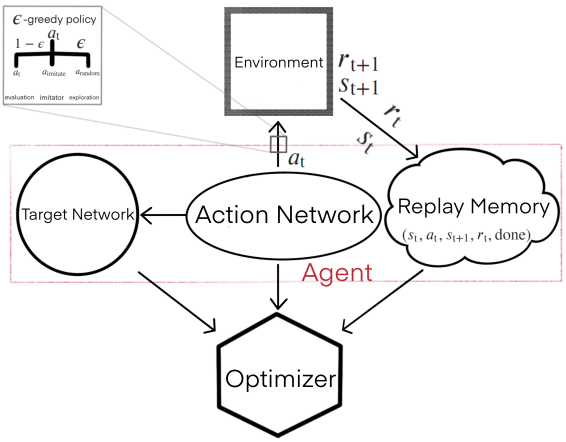


Figure 22: Final Structure of the CNN

4 Alternative attempts

Julian Aßmann

4.1 Gym environemnts

OpenAI Gym is "a toolkit for developing and comparing reinforcement learning algorithms". Specifically, it allows developers to define environments, in which agents operate, in a defined and standardized way. This general interface allows for unified benchmarks and a generalized way to communicate between arbitrary environments and agents.
Gym environments define an action space and a observation space. In our case the observation space might be the entire playing field, a two-dimensional array of integer values in a defined interval, where each value corresponds to a state of the respective square on the field (e.g. free=1, wall=2, coin=3). The action space in our case is an integer scalar, where each number represents a possible action of the agent (e.g. drop bomb=0, going up=1).
Gym environments make the assumption of discrete time step. For each time step, the *step* method takes is called. It takes an action, that the agent would like to perform in the current time step, and calculates the changes of the environment according to the action the agent executes. Other internal changes (in our case these are the other agents acting according to their programming) are also applied here. The method returns an observation in the observation space, a reward for the agent, and an indicator whether or not the game is done.
A *reset* method allows to reset the environment, e.g. before playing another round. In our case, this method initializes the playing field/map with random placement of walls, crates and the agents.

5 Looking back - Conclusion

Rhea Habermann, Daniel Richter

In this report we presented our strategies and techniques to successfully develop a reinforcement learning model which can play the game Bomberman. The two main approaches we pursued were both based on Q-Learning. While our Q-table may not have been able to sufficiently play against strong opponents, the DQN-model on the other hand played astonishingly well.
For a general introduction beginning with a Q-table was most certainly a good start. However for a Q-table to master the game very good features would be required, thus the learning aspect loses significance.
Going with a Deep Q-Network, was significantly more challenging but seemed like a more suitable approach regarding the main topic of this lecture, machine learning. Unlike the length of the respective chapters may indicate, the latter model used up substantially more time. Getting the network up and running, working with entirely new data formats like tensorflow Tensors, and primarily training took up hundreds of hours.
Undoubtedly there has to be plenty of headroom for improvement. Combining loosely better refined features with multiple networks for different game states, e.g. early game and late game seemed very promising.
Another area of improvement could be the network architecture. Something we looked into extensively but never understood in it's entirety were Recurrent Neural Networks (RNNs) and Long/Short Term Memory (LSTM) Networks. The idea was, instead of giving game states as inputs, increasing the input to sequences of game states. This should empower the network to be able to learn strategies and thus greatly improve playing strength.

Regardless of all backlashes we faced, we did learn a lot and got a more fundamental overview and general picture of reinforcement learning. We got better grasps of disparities between how we learn, understand and perceive our surrounding in compassion to NNs.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Meghashree Jl, Mirza Cilimkovic, and Vladislav Skorpil. Back propagation.

[3] Xiang Liu and Daoxiong Gong. A comparative study of a-star algorithms for search and rescue in perfect maze. 04 2011.

[4] Mazur Matt. Back propagation explained.

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. 2013. cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, and Shane Legg Demis Hassabis. Human-level control through deep reinforcement learning, 2017.

[7] John Sum, Chi-Sing Leung, and Kevin Ho. A limitation of gradient descent learning. *IEEE Transactions on Neural Networks and Learning Systems*, 31(6):2227–2232, 2020.

[8] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992.