March 29, 2021

Reinforcement Learning in an Arcade Game

Elias Olofsson, Julia Siegl

Abstract

In a competition to design the best performing agent in the classic arcade game Bomberman, we utilize a set of reinforcement learning tools and methods to achieve the goal of creating a competitive player in a student player-vs-player tournament. The final agent employs a model-based approach based on a combination of N-step Temporal Difference Q-learning with linear regression through batch Stochastic Gradient Descent on a hand-crafted feature extraction function, aided by prioritized experience replay and a variety of auxiliary rewards during training. We achieved fairly good results with our agent when playing against three other supplemented and very high-performing rule-based agents, trained for 26h on a moderately powerful laptop, where we collected on average more than half of the coins available in the game and eliminated 1-2 opponents every game, while moving efficiently and staying alive. However, we express doubts on our agent's competitiveness against other teams' agents, which may be more advanced or behave differently than the rule-based agents, which could be hard for our agent to handle. Nevertheless, we are satisfied with the general outcome, even though we could see further improvements by fine-tuning the large set hyper-parameters available or even looking into other, more advanced approaches such as deep Q-learning or other alternatives.
The full project can be accessed at Github via https://github.com/eliaserland/bomberman_rl.

Contents

1. Introduction

In this report we describe our journey while implementing a reinforcement learning model to play the arcade game Bomberman. For this purpose we followed the approach suggested by the instruction by splitting the project into three main tasks, which helped by breaking the large project into smaller pieces, which were individually easier to digest. The first sub-task was to train an agent who can navigate efficiently inside the arena to collect coins without crates or opponents. The second sub-task was to have an agent who can also win the game when crates are included. Finally, the agent should be able to play the full game with opponents and remain competitive. Even though this was the general method we worked along, this report will have the heaviest emphasis on the last and final version of our agent code. However, here we will describe shortly the general path we took to achieve this final goal, and the many approaches and variants we tried on our way.

We started with a naive and fairly simple model-free approach where we manually chose features and filled a Q-table explicitly. Since this is a rather inefficient method and training takes a long time, we later decided to change to a model-based approach where we trained a linear regression model for each action. Here we also manually chose and tuned our features by hand. However since this is rather laborious and does not generalize very well to other problems, we tried to introduce a method of semi-automatic feature extraction, utilizing e.g. Incremental Principal Component Analysis or some other method of dimensionality reduction. After realizing that we did not have time to pursue this path, we discarded this approach and instead tried to further refine and optimize the manual feature selection. We implemented both the ε-greedy algorithm and the softmax-function as decision strategies for our agent, but ended up relying on the former alternative, mostly because of time constraints which prevented us to get familiar with specific choices of good parameters for the softmax-function, which benefited the ε-greedy approach due to its lower complexity. The training of the agent model was performed online using Stochastic Batch Gradient Decent coupled with Prioritized Experience Replay for increased effective learning rate. We introduced auxiliary rewards to aid the convergence rate of the model, and manually tuned the numerical values for these rewards, which is also what we suspect where some of the major weaknesses of our trained model originates from.

Since we worked closely together in many of the approaches, we do not feel very comfortable to draw sharp lines in the sections or subsections below to divide the credit amongst us two, since most text written in this report have been spliced together with work from both of us. We can however try to explain the loose assignment of work we had between us within the scope of this project.

In the first section we go through the theory of the methods we used to train our agent. We first explain Q-learning in general and the model-free attempt in training the agent, which mainly Julia had the responsibility for. We thereafter go into the details of

our chosen model-based attempt using linear regression, where Julia formed the base while Elias optimized the approach. After this we introduce N-step Temporal Difference learning which Julia implemented and tested, mostly by herself. Following this, we have the feature selection where Julia laid the baseline in the earlier phase of manual feature design, and Elias used this work in continuation with further refinements and with the addition of prioritized experience replay which eventually gave the final breakthroughs in the project. Elias also tested and implemented the PCA and kernel PCA approach. The next section covers our training process which we arrived at very much together through our close communication during testing and evaluation of the code. We explain some key points, like the decision strategy used, the optimization strategy employed and the auxiliary rewards we introduced in greater detail. After this we conclude with our results by discussing the training efficiency and performance obtained by our final trained agent. We end the report with an outlook on how we would continue with the project if we would have more time and provide some ideas on how to improve the initial project setup for feature students.

2. Methods

The final agent we arrived at is an implementation based on N-step Temporal Difference (TD) Q-learning with a linear regression model on a hand-crafted feature extraction function, fitted via a combination of batch Stochastic Gradient Descent (SGD) with prioritized experience replay. A plethora of other methods were also tried and abandoned during the progress of the project, such as Q-table learning, Principal Component Analysis (PCA) and Kernel PCA among others, as described in [1].

Here we describe in order the essential elements to our agent, and also some of the methods considered but not employed by our final model.

2.1 Q-learning

Here we shortly recap the general Q-learning method which was introduced 1989 by Watkins [2], based on learning of the Q-function or the *action-value function*, defined as

$$Q^{\pi}(s,a) = E_{\pi}\{R_t \mid s_t = s, a_t = a\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right\}, \qquad (1)$$

where π is the given policy, s is the given game state, a is the chosen action in the first game state and γ is the discount factor. The idea is to find for every game state, the expected return of the cumulative rewards for the rest of the game, given that we take a certain action in this first state, and thereafter follow the policy at hand. If the expected return is known in every possible game state and for every permissible action, we in theory know everything there is to know about the game behaviour, and thus we can easily choose the most profitable step in any game state to obtain the optimal behaviour.

This approach is based on a model-free treatment and therefore a very common solution for reinforcement learning for games, given that the state and action spaces are relatively small. If the game and action states can be described in a discrete manner, we can utilize a table as a representation of action-value function. The update policy for an entry in the Q-table is given by

$$Q^{new}(s_t, a_t) \longleftarrow Q(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot \max_a Q(s_{t+1}, a) - Q(s_t, a_t)), \tag{2}$$

where $Q^{new}(s_t, a_t)$ describes the new value in the Q-table element of the state s and action a, at a given time t. This value is updated by the correction of the old value with the temporal difference $[(r_t + \gamma \cdot \max_a Q(s_{t+1}, a_t) - Q(s_t, a_t))]$ multiplied with the learning rate α. The temporal difference describes the different between the current value $Q(s_t, a_t)$ and the reward r_t the agent gets when performing action a_t in state s_t, added by an estimate of an optional future value $\max_a Q(s_{t+1}, a)$, which is weighted with a discount factor γ.

Using this rather straight-forward Q-table approach was initially reasonably effective in the project. When we started out with the first sub-task of letting the agent collect coins on an empty board without crates or other opposing agents, we got a good starting point by employing this method. Keeping the problem as simple as possible by reducing the number of states to a minimum, while still being able to navigate in the arena was the key point for success.

To achieve this we used the relative step difference between the agent and the closest coin in horizontal and vertical directions. When not excluding the invalid tiles, in the 17 x 17 large arena this means that an agent can have a step difference of [-15,-15] up to [15,15], resulting in 900 unique possible states and Q-table with 900 x 6 entries, given the six actions permissible in our game. Since this is a fairly small number of states, it is still very much possible to train a full Q-table by performing Q-learning as per the normal update rule presented in Eq.(2) and converge at a good result within a reasonable computational time and memory usage. Even later when we increased the number of features by a factor of three to encounter the relative position of the agent, increasing the size of the Q-table to 2700 x 6 entries, this approach can still be used with reasonable results at an acceptable computational price.

2.2 Q-function regression

However, when the state space necessarily grows in order to enable more complex behaviour in a larger environment, the Q-table approach with explicit value-estimates for every action-state combination quickly becomes unfeasible due to the so called ”curse of dimensionality”. Therefore it is necessary in most real-world reinforcement learning applications to consider model-based approaches and functional approximations of the Q-value function. The simplest functional approximation is to assume a that the Q-value

function can be modeled as a linear system, i.e. that we assume

$$Q = X \cdot \beta_a \tag{3}$$

where $X = \phi(s)$ is some given feature extraction function applied on the game state s and β is the parameter vector of the linear function for the given action a. In other words, we will fit a linear model, and thus obtain a parameter vector β, for every action separately. The benefit of using this linear assumption is that we can then formulate the problem as a regular linear regression problem, for which there exists a large amount of well-established tools, methods and preexisting knowledge. Also there are multiple libraries with implemented solvers to problems of this type which we then can utilize effectively and efficiently within the context of this project, for the sake of time management. The main drawback of this linear assumption is that the assumed linearly in X puts high demands on the quality of the feature extraction function ϕ. As such, all learning will be greatly limited by how well this feature extraction function is designed.

The normal Q-value update scheme of Eq.(2) can be formulated as a regression problem by first getting the feature matrix $X = \phi(s_t)$ from the assumed existing and static feature extraction function ϕ, and then defining the targets as

$$y_t = r_t + \gamma \cdot \max_a Q(s_{t+1}, a), \tag{4}$$

where r is the reward from the current state s at time t, with γ as the discount factor and Q is the current estimate of the Q-value function. Then the problem can be solved using any standard linear regression solver, which minimizes the squared loss as per

$$\beta_a \longleftarrow \arg\min_{\beta_a} \sum (X(s_t) \cdot \beta_a - y_t)^2, \tag{5}$$

over the entire training set, for each action a. By then playing the game and generating feature vectors X and target vectors y, we can repeatedly solve this linear regression problem to incrementally improve the model.

Practically we chose to solve this regression problem with help from Scikit-learn's library of machine learning tools. After trying out various different linear regressors we settled on the `SGDRegressor`[3], implementing the method of Stochastic Gradient Descent (SGD) for linear regression. Why we finally chose this method was because of the speed of execution for the algorithm, paired with reasonably quick convergence and its general suitable properties for large-scale problems. The implementation also had the class-method `partial_fit`[4] which allowed for more manual control and fine-tuning of the fitting process, and most importantly allowed for online training of the regression model. We also tried out the normal class-method `fit` with the parameter `warm_start` enabled which reuses the model parameters reached in the last call to `fit` for faster upstarts. However, we settled on only using `partial_fit` to do the fitting for us, since it produced the best fitting results.

Since the linear model assumption specifies that there is 6 different linear regressions occurring in parallel, one for each action, we needed a convenient way of wrapping the multiple regressors we have into a single portable unit. Sklearn provides a wrapper class named `MultiOutputRegressor`[5] which enables multi-target regression in a quick and convenient way for their selection of regressors. We used this class for a long time during the project, but we did however miss one key aspect of this class that gave us problems which was not obvious for us to detect. This wrapper class takes a given estimator, say `SGDRegressor`, and clones it internally for as many dimensions as you have in your target vector y. This does make sense for most use-cases, but not for ours however. We did want 6 parallel regressors, but since we only have one scalar target y_i per training sample X_i corresponding to a single game state transition (since there can only be a single action taken in a game transition), we could not feed it the target vector y of length 6 it demanded during training. To circumvent this problem we naively chose to estimate the 5 remaining values by using the model itself, and then appending the ”real” single target value we got through Eq.(4) to this vector, which we then fed to the wrapper regressor. This approach seemed valid at first, since if the model is good, then the estimated 5 values should be representable as well. But since we always start out with an untrained model, what actually happens instead is that in most cases we feed the regression algorithm with 1/6 ”good” points, generated by Eq.(4), and 5/6 ”bad” points which we estimated from the current fitted model. However, since we still got the model to converge somewhat using this approach, it seems like we did not fully break the training process by using `MultiOutputRegressor`, even if we most likely lowered the convergence rate by a large factor. The reason why we did not notice any major concern with this approach until quite late in phase of the project was that we had other, much larger problems at hand to solve for a long time.

To address the problems raised above regarding Scikit-learn’s given wrapper class, we implemented our own wrapper `CustomRegressor` to do exactly what we needed it to do. Here we perform almost the same tasks, cloning the given estimator into as many as we need and fitting each regressor individually with only one single scalar target per instance. Then, if we call the `predict` method, we can specify it to return an output vector of length 6, concatenating the output from each individual regressor, which was the original reason for wanting to use `MultiOutputRegressor` in the first place.

2.3 N-step Temporal Difference Learning

The Q-function regression approach presented above uses the general method of Temporal Difference (TD) Q-learning, where the Q-value function gets estimated by incremental improvements by bootstrapping the current estimate of the Q-value function, with the real rewards experienced in the current state. However, the target definition given in Eq.(4) can also be seen as a special case of the more general N-step TD learning algorithm we is explained in greater detail in [6], where the Q-value function update instead uses the real experienced rewards obtained in the N first steps after the target game state. This

means that we can improve the target definition in Eq.(4) by including the rewards r for n steps, as per

$$y_t = \sum_{t'=t}^{t+n-1} \gamma^{t'-t} r_{t'} + \gamma^n \cdot \max_a Q(s_{t+n}, a), \tag{6}$$

which reduces to what we had previously in Eq.(4) for $n = 1$. By using this N-step approach we try to mimic the expected cumulative return rewards in the game in a more accurate way.

Unfortunately we didn't see major improvements in our training when using $n >> 1$. Rather, it seemed like our training got worse and produced noise and oscillations in the training progress. We reason that this may be due to our usage of a high discount factor $\gamma \approx 1$, which may overestimate the importance and correlation of later experienced rewards with the current game state and choice of action. Through trial-and-error, we found that the model learning seemed to be most stable when using $3 \leq n \leq 5$ and a discount factor $0.75 \leq \gamma \leq 0.85$. We suspect that the reason of n being in this range might be due to the fact that the bomb counter in-game was set to 4 steps before detonation, which sort of determines the characteristic time scale of this particular Bomberman game.

2.4 Feature Selection

Since we chose to model the Q-value function with a linear function approximation as described above in this section, finding a good feature extraction function ϕ for getting the feature vector $X = \phi(s)$ from the raw game state dictionary s was arguably the most crucial part of this entire reinforcement learning process. We tried many different approaches and methods along the way, some more successful than others, but ultimately settled on a rather basic but still very powerful approach of extracting as much information from the game state dictionary as possible and presenting it in as few features as possible.

Our final agent uses a feature extraction function that outputs a vector of length 10 for every game state input. These features can mostly be sorted into two categories; information useful in a lethal situation, i.e. when there exists an imminent, lethally dangerous threat to the agent from an active ticking bomb whose blast radius is reaching the agent's current position, and information useful in a non-lethal situation, where coin collection and general movement towards crates and other opponents are the most important objectives.

With this general idea in mind, we designed the first feature to be an binary indicator telling the agent if it is currently in lethal danger. To achieve this, we loop through all the active bombs, and starting at each bomb's coordinates, we increment the position in each of the four directions until we either hit a wall, or we exceed the blast radius of 3 tiles. At every tile following this process, we check if we match the tile coordinates with the coordinates of our agent. If there is a match at any time during this process, the agent is in danger and we return a one (1). Else, if this process is fully completed for

all bombs and without a successful coordinate match, we are clear of all active bombs and we return a zero (0). This feature is important for the simple reason to make the agent aware that it cannot waste any further time and need to put itself in safety as fast as possible, else game over.

However, just an indicator for danger is not enough for the agent to know what to do to actively take itself out of danger. For this reason, we designed two features for helping with the escape from a lethal region. We tried many different ways of achieving this throughout the process of the project, but the final approach that we settled on uses a breadth-first-search to find the closest tile to the agent that would put itself in a non-lethal position. During the breadth-first-search we create a graph of all the nodes (tiles) visited in the search, and their connections to each other, which we then can use to recover the path from the agent to the tile of closest escape. This is achieved through a backwards traversal of the graph from the target node (the escape tile) to the source node (the agent's tile). The output feature we then return is a vector of length 2 which indicates the relative direction towards the very first tile on this path towards the escape tile. Thus there exists only 5 unique outputs of these 2 features, namely $(0,0)$, $(1,0)$, $(-1,0)$, $(0,1)$ and $(0,-1)$, corresponding to each of the possible movement options. The first tuple with only zeros is the default return if no escape is possible, or the agent is already standing on a non-lethal tile.

We realize that these features may be too simple and gives the agent almost too much information, leaving not too much on the table for the agent to learn by itself. However, even when using this approach, we argue that the learning process still took quite some time for the agent to handle, we even had major difficulties for a long time to get the training to converge to anything useful at all, no matter how many tricks and improvements we gave it. One thing we see now after-the-fact is that we tried almost all of these approaches, save for the final version of agent model, with the wrapper class `MultiOutputRegressor` and its associated problems mentioned above. As such, it may be the case that many of the early approaches we attempted could actually have succeeded in a much more satisfactory way if we had implemented our `CustomRegressor` at an earlier stage in the project. However, as we discovered this mistake quite late, there was no time left to go back and retry our earlier feature design attempts to see how they would compare with the new custom wrapper class.

Before we started with the backwards-traversal of the graph to recover the shortest path, we also tried to return the normalized relative position vector between the agent's position and the position of the closest escape tile, without any consideration of what the actual path to this tile was. This approach had been successful for us before when training the agent to collect coins on an empty board with no crates and no other agents, but when tried in the context of indicating the escape route direction, this method had limited performance and seemed to produce noise which would affect the movement efficiency of the agent. We reason that this is due to the fact that the optimal path to a

certain tile may not necessarily be correctly characterized by a vector giving the direction of the ”bee line” towards this tile. There may be obstacles in the way which would need to circumvented in an intelligent way, that could not possibly be represented by a single direction vector towards the end goal.

Then, for the remainder of the features, we operated in a very similar manner. When in a non-lethal situation, there are three distinctly different objectives an agent can go for; collection of coins, movement towards other agents and movement towards crates. For each of these objectives, we first used a breadth-first-search to find the nearest tile that would in the best way fulfill the objective at hand. In case of the offensive movements against other agents, we wanted to find the closest tile that would place any of the other agents in a lethal status if we laid a bomb at this location, while simultaneously allowing for our own agent to escape the explosion. For the coin objective, we searched for the closest coin that was simultaneously reachable by the agent. For the crate objective, we searched for the best crate position within a specified radius that would destroy the largest amount of crates, weighted by the number of steps required to get there. To determine which was the best crate-destroying tile, we assigned a score w to all tiles within the search radius, which gave the average number of crates destroyed per step, as per the formula

$$w = \frac{c}{4+s}, \tag{7}$$

where c is the number of crates that would get destroyed by a bomb placed at this location and s is the number of steps required to get to this location. The the number four in Eq.(7) comes from the always present 4-step bomb timer delaying the detonation after bomb placement.

When the best tile was determined for either the offensive objective, the coin objective or the crates objective, we performed the backwards-traversal of the graph we had created during the breadth-first-search, in order to recover the shortest path from the agent to the target tile. Similarly as for the escape route, we then chose to return the relative position vector from the agent's position to the very next tile on the path towards the target tile as the corresponding features for each of the objectives. Our discussion from above regarding if this is a valid approach, or if it gives the agent too much information, applies in these cases as well. We tried a multitude of other, less informative features before settling at this solution, but all of them gave bad performance with noisy and ineffective learning, or gave seemingly no convergence at all.

Finally, we needed a single additional binary indicator as the last feature, telling if the agent had arrived at the crate-destroying goal tile or the offensive objective goal tile. This was necessary since we had reserved the return $(0,0)$ for all four of the different direction vectors as the default, or ”no-response” fallback result. For the escape direction vector, $(0,0)$ implied that escape was impossible or the agent was already at a non-lethal tile. For the coin objective, the zero-vector implied that there was no reachable coin present within the search radius. However, for the crate and offensive objectives, the

zero vector could either mean that the target was reached and the agent had arrived at the correct tile, or there was no crates or opponents reachable within the search radius. Since the first case, target reached, should trigger the agent to bomb the current position, while the other case should not, we needed an additional feature to differentiate between the two cases. The solution was to create an additional feature, a binary indicator giving the agent clearance to bomb the current tile. We also added some extra complexity by only allowing the agent to attempt to drop a bomb when the current tile did not have a lethal status, and bombing was available (did not already have an active bomb placed). Also we let the agent prioritize other opponents over crates, and as such would not be given clearance to drop a bomb if any other agent was in the proximity, even if the agent was currently standing on the crate goal tile.

As an additional safety, we later added a switch to forbid all of the direction vectors for the non-lethal objectives to suggest a direction which would lead into a lethal tile. Thus if the very next tile on the path towards any of the non-lethal objectives would be detected as a lethal tile, then the return from these sub-routines would default back to the zero-vector, hopefully suggesting the agent to wait the turn, or walk in a different direction.

Thus in summary, our feature selection function returns a vector with 10 values, which comprises of two binary indicators (lethal status, target reached) and 4 different direction vectors giving the relative direction towards each of the four game objectives (escape, opponents, coins, crates). All values are in the range of $[-1, 1]$ and thus we did not need any pre-processing with normalization or standardization before feeding the features to the SGD regressor.

2.5 Incremental Principal Component Analysis

Another feature selection approach we also tried in parallel to the hand-crafted approach we presented in the preceding subsection, is a ”semi-automatic” feature reduction method, complementary to the manual approach of crafting your own features. The general idea was that the general agent model would train for a specified time period with a static feature extraction function, during which the information of all (or a subset of) the experienced game states would be saved in a large list. When it was deemed that enough data had been collected, the agent model training would stop and the saved game state history would be used to fit a dimension reduction model of choice. When this fit was completed, the agent model would have to restart its training from scratch, since the underlying feature extraction function (which was assumed to be static) now had changed and made the trained model parameters out-of-date. This cycle of alternating learning of the agent model and the feature extraction function could in theory continue until convergence. The hope was that agent model training would gradually improve and be made easier as the feature extraction function would periodically be refined and perfected.

However, we only got to implementing the base structure required for such a training

scheme and testing the functionality using a very rudimentary and unintelligent feature extraction function (which converted the raw game state dictionary to a vector of length ~ 300) in combination with Scikit-learn's `IncrementalPCA` [7], an implementation of Principal Component Analysis (PCA) that enables incremental learning. This expectedly produced very poor results, due to the fact that PCA is only a linear dimensionality reduction method, while the feature extraction function we fed it most likely is highly non-linear. However, we believe that the usage of a incremental dimensionality reduction method is essential for this application due to the online nature of the problem, since the set of training data in not known from the beginning, but rather arrives continuously as we play the game and we want to gradually improve the fit as more data becomes available. Also, using an incremental algorithm should be beneficial whenever we have very large sample sizes, since we do not have to load all training instances into memory at the same time.

The problem is that we have to contend with the non-linearity of the features in the game state dictionary in some clever way. One option is to continue on the same general path as described in the preceding subsection, by hand-crafting a underlying feature extraction function which is good enough that the application of PCA to it produces some good and useful results. The other option is to look into non-linear dimensionality reduction methods, for example the related method of Kernel PCA. The caveat with this approach in particular is that Kernel PCA in its standard formulation requires that all training samples be kept in memory, due to the fact that they need to be accessed at the transformation stage, when we want to apply the learned feature reduction model to new, unseen data. Also due to the fact that the Gram matrix will be of size $N \times N$, where N is the number of training instances, the computational complexity for the transformation and the memory requirements for storing the Gram matrix quickly becomes prohibitively large for even moderately sized training sets. One way of dealing with this would be to look into some method of cluster analysis to convert a large training set to a smaller one by only considering cluster representatives. Another option could be to look into incremental versions of Kernel PCA, e.g. the method suggested by F. Hallgren [8].

However, when we got to this stage in the project, there was not enough time left to seriously venture into all of these options in a satisfactory way, and as such we had to make a choice of which line to continue with. Ultimately, we made the choice to abandon this automatic feature reduction approach and instead focus our efforts into the hand-crafted feature extraction approach described in the previous section. Thus, the remnants of this automatic feature extraction attempt are still present in our final version of the agent code, but these methods are not utilized in any meaningful way in our final agent.

3. Training process

The final version of our agent utilizes a variety of different methods during the training process to aid the learning efficiency and increase the speed of convergence for the model. In summary we use the ε-greedy decision strategy for balancing between exploration and exploitation, Stochastic Gradient Descent on a randomly selected batch from the saved transition history, optionally used together with Prioritized Experience Replay to reduce the batch size by keeping the most significant training instances. We also employ a set of auxiliary rewards in addition to the canonical rewards defined by the game logic, as to give guidance to the agent which helps to improve the rate of convergence.

3.1 Decision strategies: ε-greedy and softmax

In our final version, we have two kinds of decision strategies implemented; the ε-greedy algorithm and the softmax function. The purpose of both of these methods is to convert the output vector from the Q-value function model to a specific choice of action for the agent, given the current game state. Since the Q-value function model estimates the expected cumulative return of rewards for the given state-action combination, we have a vector of length six (one estimate per permissible action), that we need to translate into probabilities for selecting each specific action. The ε-greedy algorithm is quite straight forward as it uses an additional scalar number $\varepsilon \in [0,1]$ as a threshold to select between one of two selection approaches. Every time an action is supposed to be taken, a random number is drawn from a uniform distribution between 0 and 1. If the drawn number is below the threshold, we choose an action uniformly at random from all permissible actions in the current game state. In the other case, we take the single action that corresponds to the highest Q-value, i.e. we apply the argmax-function to the Q-value vector. This approach is easy to understand and implement, and solves the problem of balancing between exploration and exploitation in a simple, straight-forward way. It is also the method we chose to employ in the training process of our final agent.

However, we also tried out an alternative method to this approach, which possibly can intuitively be seen as a more intelligent method of choosing an appropriate action for the current game state. The softmax function uses the Maxwell-Boltzmann distribution originally found in statistical mechanics, and can be seen as a way to get probabilities for all actions, weighed by their respective Q-values, as per

$$P_t(a) = \frac{\exp\left(Q_t(a)/\tau\right)}{\sum_{i=1}^{n} \exp\left(Q_t(i)/\tau\right)}, \tag{8}$$

where n is the number of possible actions in the game state. Here there is an additional parameter $\tau \in (0,\infty)$ that is typically called the ”temperature” in analog to statistical mechanics, and provides the key functionality of this function. Depending on the temperature, we smoothly transitions between two limiting cases. When $\tau \to 0$ we recover the argmax-function we used in the ε-greedy algorithm, and when $\tau \to \infty$ we

also recover the uniform distribution for all actions. Thus the softmax-distribution is somewhat similar to the ε-greedy algorithm, but can be seen as a less aggressive alternative to the quite harsh decision strategy that is the ε-greedy algorithm. With the softmax function, many similarly good options for actions can be considered somewhat equal proportions due to the relative weighing of probabilities, which is details that gets lost in the ε-greedy treatment. This is also where the name "softmax" comes from, since it can be seen as a "soft" argmax-function, depending on the value of the parameter τ.

Practically we also had to implement a sort of normalization for the softmax function in our code due to the limitations of floating point numbers. To ensure numerical stability, what we actually implemented as the softmax function was the function

$$P_t(a) = \frac{\exp\left(Q_t(a)/\tau - \max(Q_t(i)/\tau)\right)}{\sum_{i=1}^{n} \exp\left(Q_t(i)/\tau - \max(Q_t(i)/\tau)\right)}, \tag{9}$$

which can be proven to be mathematically identical to Eq.(8) using the rules of exponentials, but is much less prone to numerical over- and under-flows in practice.

Even though we have implemented the softmax-function as a complimentary decision strategy to the ε-greedy algorithm, we chose to not actually use this method for the training of our final agent, mostly due to the fact that time constraints did not allow us to get familiar enough with the specific numerical values appropriate for efficient training. At the same time, ε-greedy requires less specific knowledge and can helpfully remove complexity in the implementation during many of the intense phases of debugging we encountered along the way.

Also during the training phase, we gradually anneal the exploration rate ε as the number of played games increases. For our final agent, we used an initial ε of 1.0 with a decay rate of 0.99995 applied to ε at the end of every game. These values seemed to give the most stable learning behaviour for the final learning model and the hyper-parameters chosen.

3.2 Batch Stochastic Gradient Descent

Since we utilized the linear regression method of Stochastic Gradient Descent during the model fitting stage of the learning process, we had to decide on a approach of how to feed the training instances to the regression algorithm. The method we settled on uses a batch-approach of randomly selecting a subset of the training instances available in the transition history. This seemed to help against over-fitting, since the the random selection prevented that many subsequent transitions, which would be heavily correlated, weighed in too heavily in the fit. This however required that the chosen size of the batch was a relatively small fraction of the total amount of the available transitions saved in the history.

We also experimented with the max length of the transition history list, i.e. the limit of how many of the most recent transition tuples to keep in memory. Through trial-and-error, we discovered a bit counter-intuitively that the learning process seemed most efficient

when we used a short transition history together with a rather small random batch size. We argue that the reason for faster learning convergence when using a smaller transition history is due to the fact that the reinforcement learning cycle, i.e. the feedback loop of action and consequence, becomes shorter and more agile with a shorter history size. Thus the previous actions of an older version of the agent does not linger in the memory for as long, and the forwards-progression of the learning becomes better. Specifically, we settled on a transition history with a max length of 1000 entries, with the size of the random batch to half of this size. When specifying the training frequency to one call to `partial_fit` at the end of every game, we saw quite rapid learning and a steady improvement without major oscillations or instabilities. Thus, this is also the parameters used in the training of our final agent model.

3.3 Prioritized Experience Replay

Another improvement to the random batch selection presented in the preceding section is Prioritized Experience Replay, which we added as an compliment to increase the effective learning rate of the agent model in our implementation. In Prioritized Experience Replay, the idea is to calculate the residual in relation to the current version of the fitted model, for every instance in the training set and only keep a subset with the largest squared residuals. For a training set with feature vectors X_i and targets y_i with corresponding action a, the residuals r_i are defined as

$$r_i = y_i - Q(X_i, a), \tag{10}$$

where Q is the Q-value model that is assumed to exist with a prior fit. Calculating the squared residual for all instances in the training set, we can create a subset of the training set, which contains the feature vectors and the targets of the instances with the largest squared residuals. These instances will have the larges impact on the regression fit and thus are the most effective instances to include from the training set. Conversely, instances with residuals closer to zero are already well represented by the previous model fit, and thus they can largely be omitted from the training set for increased computational efficiency.

In our final agent model, we specified the relative size of the subset generated from Prioritized Experience Replay to be 0.25 of the size of the randomly selected batch mentioned in the previous subsection. This combination of parameters seemed to yield good and stable results for us, without any major oscillations or other unwanted artifacts in the learning process.

3.4 Auxiliary rewards

Perhaps one of the most essential tools we had at our disposal for increasing the convergence rate of the learning process was reward shaping, through the introduction of fictional game rewards which would guide the agent in a helpful way. This was in general

a very difficult task for us to achieve. To encourage the agent to perform well on the goal objectives of the game, while not leaving any loop-holes that the agent could exploit in an unwanted or unexpected manner, was not easy. We tried many approaches that seemed to be air-tight in the design phase but proved to be ineffective, or even directly harmful to the agent, in practice.

The final set of auxiliary rewards we settled on can be grouped into two main categories, similarly to how we made the design choices for the manual feature extraction function described in the previous section. Firstly, we have a set of rewards when the agent has a lethal status, i.e. that there exists an active ticking bomb which will at detonation reach the agent at its current position. Thus, the only priority when the agent is in a lethal situation is to move such that the agent no longer has a lethal status, else we will reach game over very soon. This means that we want to reward the agent when it moves in the correct direction of escape, given by the corresponding direction vector we implemented in the feature extraction function, and punish the agent for taking any other action. It may seem like a detail to also give negative rewards for the wrong behaviour, but it is actually essential and incredibly important that the given punishment is equal in magnitude and opposite in sign to the given reward for correct movement. Otherwise, if the escape reward/punishment is not anti-symmetric for movement, the agent may accumulate a net positive reward for walking back and forth between the same two tiles. This would be a potential loop-hole that the agent could get stuck in and learn unwanted behaviour which only is an artifact of a poor design choice of the fictional rewards.

The second category concerns rewards given to the agent when it is not in a lethal situation. Within this group, we give specific rewards/penalties depending on which action was taken by the agent in this game state. If the agent had chosen to place a bomb, we check whether or not the ”target reached”-indicator was signaling or not. If the agent was standing at the correct bombing goal tile and placed a bomb, we award it with a positive reward. If the target was not reached but the agent placed a bomb anyway, we punish it with a negative reward. For the negative rewards, we found that we had to specify it to be equal to (or less than) 4 times the magnitude of escape movement rewards, to prevent loops where the agent would lay bombs alternatingly between two tiles and consequently gain positive rewards solely from repeatedly escaping and surviving its own bombs.

For the action of waiting, we had to similarly determine whether or not the chosen action to wait was warranted or not. The way we chose to accomplish this was by rewarding the agent for waiting if all four of the direction vectors we had in the feature vector was equal to the zero-vector. Since we had defined all of these direction vectors to return the zero-vector if no target objective could be found, or that there was a lethal tile blocking its path, it made sense to reward the agent for waiting in such a situation. Conversely, if the agent chose to wait while not all of the direction vectors where equal to zero, we punish the agent for waiting unnecessarily.

Finally, if the agent had chosen to move in any of the four directions, we needed to checked whether this movement corresponded to approaching any of the non-lethal objectives. To make everything less noisy and focus on whats important, we ranked the three non-lethal objectives in decreasing order of importance as; movement towards offensive tiles against other agents, movement towards coins, movement towards optimal crate-destroying tiles. For the handout of rewards, we then checked each of these objectives in order of importance, and *only* considered the highest ranked objective which had an existing target, i.e. had not returned the zero-vector. If the agent had moved towards the highest ranked objective, it was given a positive reward, else it got a negative reward with the same magnitude. Similarly to our previous discussion, movement rewards almost always have to be anti-symmetric as not to create infinite loops of net-positive rewards for the agent to get stuck in.

Another key part to the handout of rewards, which is intimately tied to the feature design, is that we have disconnected the handout of rewards from the next game state, i.e. the game state after the target game state with its corresponding action. We figured this out quite late, and it helped a lot by decreasing oscillations and erratic behaviour by the agent. By only letting the auxiliary rewards be dependent on the target game state and the action the agent took in this state, the results and the training progress got way better and smoother. We figure that the reason for this was that having auxiliary rewards be based on some difference between two game states, which was how we did it before, puts high demands on the quality and the stability of the designed features. If the ”best” objective changes between two consecutive game states, you might get jumps and discontinuities in your features which may in turn brake the functionality of your auxiliary rewards in ways that are hard to predict and prevent. Before we made this change we had large problems trying to prevent loops from occurring by the agent, and it almost seemed like every type of prevention through some additional auxiliary reward only prolonged or changed the character of the loop. However, after making this crucial change, we had no major problems with loops any longer.

3.5 Filtering of invalid actions

Another tool we used for the longest time during the training process of the agent was a function that acted as a filter which only allowed valid actions to be executed by the agent. At first this function only took care of truly invalid movement, i.e. not allowing the agent to run into walls or crates etc. and not allowing it to place bombs while having one placed down already. However, later we also added functionality for preventing ”stupid” moves as well, like prevention against bomb-dropping while standing at an inescapable tile and movement onto tiles with a lethal status. But most interestingly is that we thought that this filter would be needed at all times for the agent, as experiences from the earlier stages of the project had shown us, but in the end for the final version of the trained agent, there was no need for this filter at all. The training could be performed with or without the filter, and the trained agent’s performance would be almost identical.

As such, we chose to turn these ”training wheels” off when we trained the final version of our agent, since by letting the agent go fully free of its leash felt less like cheating.

4. Results

Here we present some of the experimental results obtained during the phase of the project. We start with the final version of our agent, the version we supplied for the final submission and competition. This specific training session lasted over 24h and involved the play against 3 other rule based agents for approximately 95000 games. The statistics generated during the training run can be seen in Fig.(1). Here we can see how the the agent quite rapidly learns how to handle the bombs to such an extent that it rather efficiently can destroy a large amount of crates per game consistently, as well as collect a majority of the coins while getting on average 1-2 kills per game. Granted, our agent is not yet any terminator which completely annihilates the other rule-based agents every game, but it is still doing quite well on average.

We can note that running this particular training session for as long as we did was not necessary, since the training had already converged by $\sim 75000 - 80000$ played games, and the last 4-6h of training did not ultimately improve the end result.

We can also see how the in-game score, which is the sum of real and auxiliary rewards, gets noisy towards the latter half of the training process, with large spikes into negative scores occasionally. We are not entirely sure of what the precise reason for this is, but what we typically experienced during the duration of the project was that these types of spikes and large oscillations may be caused by the agent finding some loop-hole in the auxiliary rewards, which creates instabilities in the learning process. However our statements from before still applies; these results are among the better we obtained during the entire duration of the project. If there was still time left, we would however like to revisit the entire setup with auxiliary rewards and try to pinpoint the reasons for the heavy oscillations, in an effort to try and prevent them.

One more reason which might explain the oscillations, which we also found from experience during the project, is that the the specific combination of transition history size and batch size, coupled with the training frequency, might cause regular over-fitting in the later stages of the training process. Since if the linear regressor gets fed almost the same sets of samples, due to the fact that instances occurring close to each other in time within the same game might be heavily correlated, the regressor might transition from an initially good learning progression into suddenly getting steadily worse game scores and other game quantities. To solve this, we would have to spend more time fine-tuning the sizes of the transition history, the random batch and the prioritized experience replay batch and which training frequencies which together yields good performance. Also, the regularization parameter α in `SGDRegressor` would have to be fine-tuned for the specific circumstances to optimize the overall performance and prevent over-fitting. Here, we did not have time to do any deeper analysis into the fine-tuning of all of these

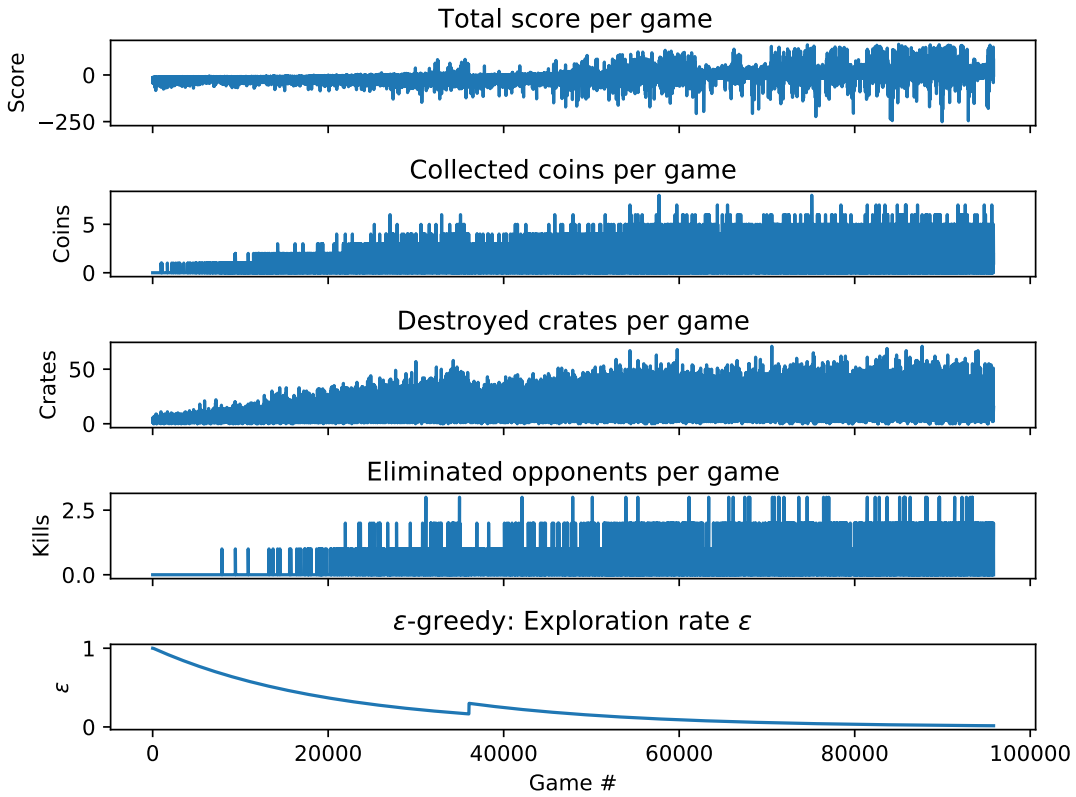

Figure 1. Training statistics from the training run of our final agent. From top to bottom; the total score gained per game (real + auxiliary rewards), the total amount of collected coins per game, the total number of destroyed crates per game, the total number of opponents eliminated per game, and the value of ε determining the exploration rate in the game. Total training time was approximately 26h. The reason for the discontinuity in the ε-graph is due to the fact that this run is comprised of two consecutive training-runs, and the program was restarted at $\sim$ 37000 played games. This training session was done without the filtering of invalid actions.

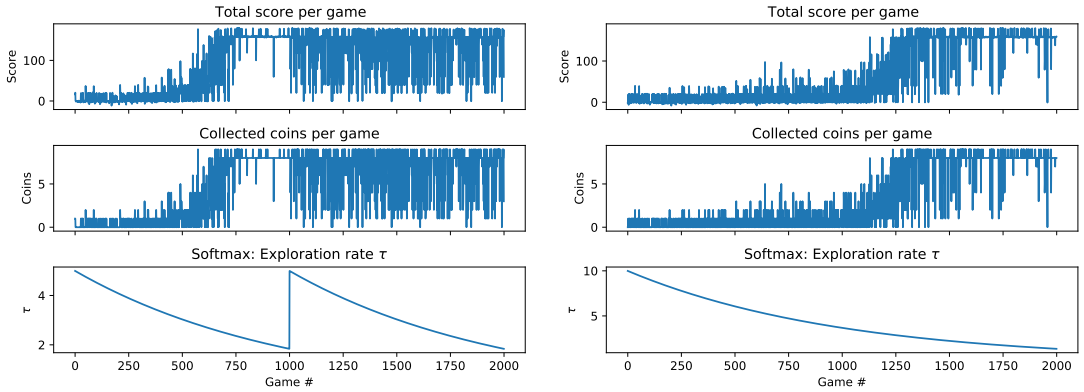

Figure 2. Training progress of an early version of the agent, collecting coins on a board without crates or other agents. In the left graph, we anneal the exploration rate and reset it mid-way in the training process. In the right graph, we simply keep a steady decaying exploration rate throughout the entire training run.

hyperparameters, since re-running these long training runs was not going to work given the time allocated for the project. The parameters chosen in the final version of the agent simply were the first values we used when we finally started to get somewhat good performance in the late stage of the project.

Another interesting observation we discovered in the early phase in the project was which strategy to use when annealing the exploration rate during training. Initially we had ideas that the annealing process might have to be restarted during the training process, in order to prevent getting stuck in some bad local optimum. Thus we tried multiple approaches of either using slowly decaying exploration rate, or using a faster decaying rate but restarting it during the training phase. We tried these experiments on many occasions in a variety of configurations, but the essence of the results can be presented by the two graphs in Fig.(2). Here we can see that we do not necessarily improve the model fit by resetting the exploration rate mid-training. Instead we found that it was beneficial to choose a rate of decay which was slow enough to in most cases guarantee that the training process would see enough randomness in the initial phase of the training and then steadily transition into heavier exploitation in the later stages. Also, choosing a very slow rate of decay seemed to help prevent oscillations in the learning progress as well.

4.1 Conclusion

In summary, we have achieved an agent model which, when trained properly, can perform almost on par with the rule-based agents. We did not obtain the same level of consistency in results as these rule-based agents, but our agent still managed to play the game in a largely successful manner. All three sub-tasks within this project; collecting coins on an empty board, destroying crates and collecting coins without other agents, and playing the full game competitively, can arguably be considered to have been achieved. However,

there is still lots of room for improvement during play against other agents, as is the case in all competitive games. And since we did not have time to train our agent against any other opponents than the rule-based agents, no self-play or play against other agents from other teams in the contest, there is a large uncertainty in how our agent will perform against more advanced opponents. Even opponents which may not necessarily play better, but have a different play-style than the rule-based agents, may give our agents a lot of trouble. But as with any arms race, which this effectively is, there will always remain space for improvements - it is just a matter of time and budget, and a bit of luck, which ones will end up on top.

5. Outlook

5.1 How to continue

As can be seen in the preceding section we ended up with some rather decent results. However, there are still many things left which we were not able to test and optimize due to the lack of time.

One large part that could have seen more improvements is the fine-tuning of all hyper-parameters present in our agent code. We especially have in mind the effective learning rates, batch sizes and regularization for our trained model, the parameters inside the decision strategies and the auxiliary rewards. We only tested a small set of parameters in the beginning and then had to stick with some of these in other to progress with the project and to further develop our approaches. Thus, exploring other combinations of these parameters may lead to a better performing agent or a speed up in convergence for the learning process. Also for the decision strategies, we only tested few values for e.g. our ε rate of decay and tried to tune everything manually to the best of our extent. It would have been nice to have figured out a more sophisticated approach in choosing these parameters than just brute-force trying every combination manually and hoping for an improvement.

Furthermore, we planned to have a closer look into a more robust or even automated creation and selection of auxiliary rewards. Currently we tune them manually which is laborious and not efficient at all. We could have introduced some sorts of reward potentials and from this on further optimize some kind of a reward function. We think that this is could potentially yield some improvements to our performance.

Also, since we run our training pretty much in a single thread on the CPU, we probably could see some speed up in execution time by trying to parallelize the training processes. However, since our agent converges within a somewhat reasonable period of time, there was no real incentive to try to convert the entire game logic and our agent code into a more efficient multithreaded version, given the time allocated for this project. Another thing we did not try, mainly because neither of us had access to any powerful GPU:s, is to improve the speed of execution through the usage of CUDA in the training. However, since we neither had the hardware nor the time, we did not consider this option

for very long.

Another thing that we would have liked to spend more time in is testing completely other approaches to the reinforcement learning. We mainly focused first on simple Q-learning and than stuck with the simple linear regression model. It would have been interesting to also train and test e.g. a deep Q-learning approach or a random forest. It would then be very interesting to compare how different the performance and the training speed would be for the other approaches.

5.2 Improvements of the setup

In general, we got along fairly well with the whole final project. We felt that the specification was detailed enough, with the single exception being that the description on the report outline could have been done in more detail. Also, the requested format of the report, which should detail the final version of the agent, was a bit difficult for us since it was very difficult to start the writing of the report while the project was not yet finished. Thus we were a bit stressed to finish the coding towards the end of the project, such that we could start with the writing (and not having to totally scrap the writing on some approaches we later abandoned).

Other reflections was that we felt in the beginning that the whole setup was quite overwhelming and it took us a long time to just organize ourselves and fundamentally understand the theoretical concepts we needed, before actually getting started on the project. A suggestion for future improvement may be a tutorial or a short tour through the existing code base in the final lecture, or in an extra tutorial, which would be incredibly helpful to get started more smoothly.

Otherwise, the templates found in the setup were quite helpful, e.g. the function `state_to_features`, but other things caused some confusion, like the `bomb_map` array in the game state dictionary, which may even by faulty. One could improve these things by providing more detailed and up-to-date documentation.

References

[1] Sutton, R. and Barto, A., 2020. Reinforcement learning. 2nd ed. [online] Available at: http://incompleteideas.net/book/RLbook2020.pdf [Accessed 29 March 2021].

[2] Watkins, C.J.C.H., Learning from delayed rewards. PhD Thesis, University of Cambridge, England (1989)

[3] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDRegressor.html

[4] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDRegressor.html#sklearn.linear_model.SGDRegressor.partial_fit

[5] https://scikit-learn.org/stable/modules/generated/sklearn.multioutput.MultiOutputRegressor.html

[6] Richard S. Sutton, TD Models: Modeling the World at a Mixture of Time Scales, Machine Learning Proceedings 1995, Pages 531-539, (1995)

[7] https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.IncrementalPCA.html

[8] Hallgren, F. and Northrop, P., 2018. Incremental kernel PCA and the Nyström method. [online] Available at: https://arxiv.org/abs/1802.00043 [Accessed 29 March 2021].