Fundamentals of Machine Learning

Final Project

Applying Reinforcement Learning to Bomberman

Submitted By :
Hein-Erik Schnell
Karl Thyssen

https://github.com/nessyht/FOML-Bomberman

Contents

1 Learning Method and Regression Model

Hein-Erik Schnell

This section is divided into the three crucial tasks for which we needed to develop a concept in order to get the agent into training. Those tasks were:

- Choosing a suitable *state representation* to be then passed to an *regressor* to estimate the expected *reward* for each of the possible actions
- Choosing suitable *rewards* in order to communicate the goals of the game to the agent
- Choosing a suitable *regressor* which estimates the expected *reward* for each possible action at the current state of the game.

1.1 State representation

Hein-Erik Schnell

The first task was to choose a suitable state representation. In case of regressors provided by *scikit-learn*, the training data is usually passed to the regressor as a 2D-array where each row (first index) represents a single state and each column (second index) represents a feature of the respective states. Analagously, the prediction then demands an array of similar form. The regressor then returns an array with as many predicted values as there were rows (states) in the input array. If one wants to predict only a single value, one may not pass a 1D-array to the regressor but create an additional dummy dimension. All this means that if we want to use precoded regressors from scikit-learn, we need to find a 1D-array representation for a single state.

After each step, the relevant data is passed to the agent via the dictionary `self.game_state`. In the agents `callbacks.py` we defined the function `create_state_vector(`*`self`*`)` which turns the information provided by `self.game_state` into a 1D-array. We chose to store the relevant features in the following way:

State representation 1

- For each cell:
  - **Agent, Opponent, None** $\{1, -1, 0\}$:
    The dictionary provides the entry `self.game_state['arena']` which is a 2D representation of the game board. We use this to create a numpy-array `agent_state` of the same shape which is 1 on the agents position,

-1 on cells with an opponent and 0 on all other cells.

- **Crate, Coin, Empty/Wall** $\{-1, 1, 0\}$:
  A copy of `self.game_state['arena']` is manipulated in such a way that it is -1 for a crate, 1 for a coin on the respective cell and 0 in all other cases. This would mean that the agent could not distinguish between empty cells and walls. This issue is resolved later when we delete all cells which contain walls. These cells are always the same and therefore do not contribute to the learning process. In our code, this whole part is represented by the variable `loot_state`.
- **Bombs** $\{6, 5 \dots 2, 1, 0\}$:
  The variable `bomb_state` is of the same shape as `self.game_state['arena']`. All cells are by default 6. If the cell will soon be affected by a bombs explosion, the values $5 \dots 2$ represent the 4-time-steps countdown. $1 \dots 0$ represent the 2-time-steps explosion. This way, `bomb_state` provides a danger level for each cell. 6 means no danger at all.

- Just once (implemented in our code as `extras`):
  - **Current step number** $\{1, \dots, 400\}$:
    `extras[0]` contains the current time step.
  - **Danger level** $\{0, \dots, 6\}$:
    `extras[1]` represents the danger level on the agents current position. It is calculated by $6 -$ `bomb_state[x,y]`, where `x` and `y` are the coordinates of the agents position. Consequently, this danger level in invers to the danger level in `bomb_state`, i.e. 0 means no danger, $1 \dots 4$ means increasing danger and $5 \dots 6$ would be bombs exploding. The least point is rather irrelevant since the agent would already have been deleted by the environment.
  - **Bomb action possible** $\{0, 1\}$:
    `extras[2]` is 1 if the agent could place a bomb and 0 if not (i.e. if an own bomb is still ticking).
  - **Touching opponent** $\{0, 1\}$:
    `extras[3]` is 1 if an opponent is on a neighbouring cell and 0 if not.

After manipulating the data in the described way, all cells containing walls are deleted from the 2D-arrays `agent_state`, `loot_state` and `bomb_state`. As already described above, this is done because these entries will always be the same and therefore never contribute to the learning of the agent. The three arrays are then flattened and concatenated after one another into the 1D-array `vector`. Finally,

we append the `extras` to the `vector`, which is then returned by the function `create_state_vector`.

With the described representation of a state we combined features which could be represented as seperate features into single features. For example in [2], each cell has a feature *Agent on cell?* and *Opponent on cell?* which can both assume the values 0 and 1. We combined these two features. As we see it, a proper regressor should be able to determine the important features as well as the relevant range of values of a feature. A *Random Forest Regressor*, for instance, should theoretically be able to do so since the way it works is to find the most relevant feature and its most divisive value.

With *state representation 1* (532 features), the feature space has at most about 5.4×10^{320} possible states.

State representation 2

As described later in section 3, our agent never managed to get out of its starting corner. In order to change that, we condensed the above state vector (532 features) into a much smaller state vector of 180 features. The main idea of this smaller state vector is that agents, opponents, crates and coins can never occupy the same cell. Bombs could occupy the same cell as agents, opponents or coins, but for the sake of the agent they shouldn't:

- For each cell:
  - **Empty, agent, opponent, crate, coin, danger level** $\{0, 1, 2, 4, 5 \dots 11\}$: Empty cells are 0. Cells with agent, opponent, crate or coin are $\{1, 2, 3, 4\}$, respectively. The values $\{5 \dots 11\}$ represent the danger level because there is a bomb affecting the respective cell. 5 is not used, $\{6 \dots 9\}$ is the four-time-steps countdown, $\{10, 11\}$ the two-time-steps explosion. If there is a danger level the values $\{1, 2, 3, 4\}$ are overwritten. This means that the position of the agent might not be represented anymore in this feature.
- Just once (implemented in our code as `extras`)
  - **Danger level** $\{6 \dots 11\}$:
    `extras[0]` is the danger level at the position of the agent.
  - **Bomb action possible** $\{0, 1\}$:
    `extras[1]` is 1 if the agent could place a bomb and 0 if not (i.e. if an own bomb is still ticking).

  - **Touching opponent** $\{0, 1\}$:
    `extras[2]` is 1 if an opponent is on a neighbouring cell and 0 if not.
  - **Position of agent** $\{18 \dots 271\}$:
    `extras[3]` is essentially the enumerated cell number of the agents position. It is calculated by $17x + y$, where x and y are the agents x- and y-coordinates.

With *state representation 2* (180 features) it has at most about 3.4×10^{193} possible states, which is much less than with *state representation 1*. For both representations, these are only upper limits. But both are calculated the same way so that these values are useful for comparisons between both representations.

1.2 Rewards

Karl Thyssen

The individual rewards for events that occur in a given step contribute to the reward function and are explicitly defined in the `reward_update()` and `end_of_episode` functions. The other parameter to influence is the discount factor γ for the discounted long term reward. We ran training with factors $\gamma = 1.0$ and $\gamma = 0.9$. The reward function is describe by a series of `if` clauses that determine the quality of the action taken and apply the reward.

- **Valid move**: -1
  Every step is punished in a small way to encourage the agent to complete tasks in the shortest possible time to avoid stacking negative rewards.
- **Wait**: -5
  We noticed while running preliminary quick trains that the agent quickly regressed into a pattern of repeated 'Waits' after only a few generations. Arguably other factors were also different at the time so its debatable how effective this now is.
- **Invalid action**: -100
  Invalid actions should be punished so the agent quickly learns the connections in between the fields and the restrictions on the frequency of bomb placements.
- **Destroy crate**: $+10$
  The agent should learn to destroy crates
- **Coin found**: $+20$
  The agent should want to destroy as many crates as possible to find coins

- **Collect coin**: +2000
  The agent is heavily rewarded for finding coins to incentivise coin collection as a primary goal
- **Killed opponent**: +10000
  If the agent does accidentally kill an agent in training it should have a large weighting so this behaviour can be encouraged, particularly as this will be rare initially
- **Die (from opponent bomb)**: −2000
  The agent should learn to avoid all bombs...
- **Die (suicide)**: −1500
  ...but rather die to its own to deny opponents 5 points.

We did not change the rewards in our tweaking due to the fact that we were not sure how to effectively test changes. Our superficial tests (anecdotal) never lead to ground breaking changes and as such we couldn't realistically change them with the expectation that something would be positively influenced. Particularly reflecting the amount of time it would have taken to train even 10 generations of the changes however some interesting ideas we would perhaps have next investigated are:

- Whether a movement to a cell should be rewarded based on proximity to other agents/bombs or coins and how we allow these factors to influence the positivity or negativity of an action. I.e. A step towards a coin that is also a step into an exploding bomb should lead to a negative reward.
- Should we punish the repetition of actions, such as the movement to a previously occupied square which may be undesirable and therefore punished with a small negative value, or should this only be the case when not running from a bomb
- Not punishing death or any actions we perceive to be negative but rather only enforce good ones and let the agent decide whether the risks are worth taking that is to say whether the punishment of not getting the reward is discouragement enough

1.3 Regressors

Hein-Erik Schnell

In order to estimate the expected rewards properly we need to find a regressor which is both flexible and very decisive with respect to the features. Flexible, because a simple linear regressor would not be able to resemble the volume of states and

outcomes. This big variety of possible states, even at the very beginning of an episode, means that many features are not relevant in a given situation. This is the reason why the regressor should also be able to find the most decisive features.

Since we will not know whether the competition will be split into two leagues, one for neural networks and one for classical regressors, we initially decided to train both a classical regressor and a neural network and see which one performs better (in its respective league).

As a classical model we chose to use a *Random Forest Regressor*. The advantage of this regressor is that it is infinitely flexible can and can cope with whatever shape the reward function may assume. However, it turns out that this regressor is also quite inaccurate and erratic. Its erratic nature might be problematic because the order of the estimated rewards, i.e. which move is the best, is very important. If the order can not be predicted correctly the agent will not choose the best but only a good move. Figure 2 shows how likely this is. We chose to use the Random Forest Regressor provided by *scikit-learn*.

As a neural network we chose the *MLP-Regressor* provided by *scikit-learn* mainly for two reasons. The first reason is that we need a precoded regressor since neural networks haven't been subject to the lecture yet. The other reason is that the syntax and available functions of that regressor are the same as for the Random Forest Regressor. This way we only need to change the initialization of our regressor and do not have to change the rest of the code. It is basically changing one line of the code in order to switch between both regressors.

The two regressors are initialized as follows:

- `RandomForestRegressor()`
- `MLPRegressor(max_iter=500)`

Thus, we used the Forest with its default settings and changed the maximum number of iterations of the MLP to 500 (default is 200).

In order to get a feeling for the performance of both regressors, we used the function `sklearn.datasets.make_regression` to create 10000 samples with 100 features of which 10 are informative. We then trained the Random Forest Regressor and the MLP-Regressor with 80% of the generated data and let both regressors predict the values of the remaining 20% of the data. Figure 1 shows how much more accurate the MLP-regressor performs compared to the Random Forest Regressor. Figure 2 shows how the MLP-Regressor maintains the order of the instances much better than the Random Forest Regressor. In case of the latter, many values are sorted into completely other regions than where they actually belong. This is also

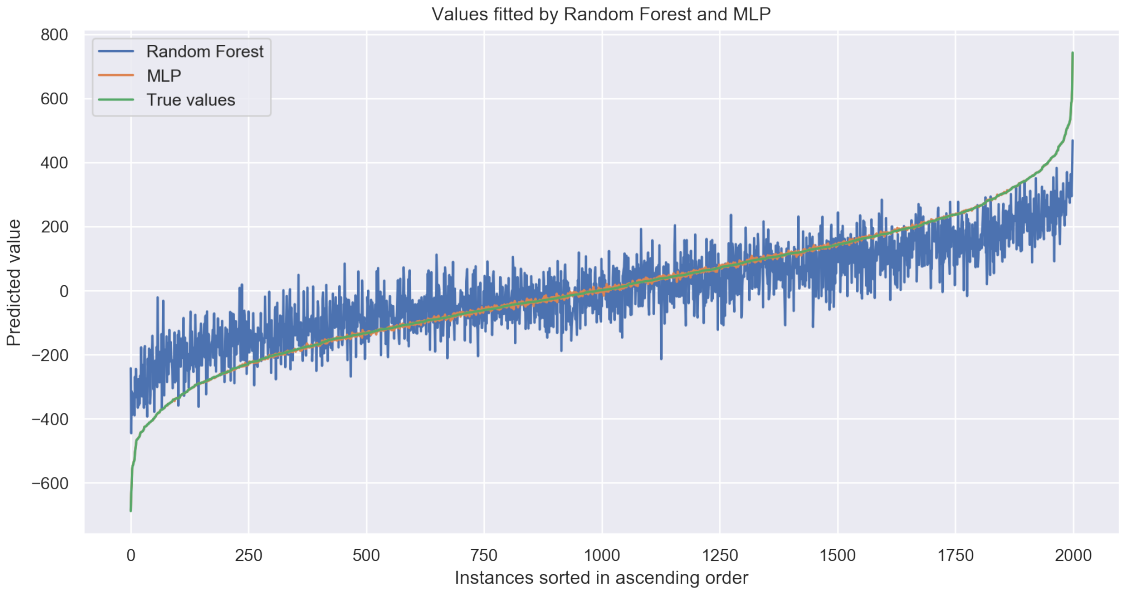

Figure 1: *For this figure,* `sklearn.datasets.make_regression` *was used to create* 10000 *samples with* 100 *features of which* 10 *are informative. We then trained the Random Forest Regressor and the MLP-Regressor provided by* scikit-learn *with* 80% *of the generated data. The figure shows the predicted values of the remaining* 20% *of the data which were used as a test set. The test set has previously been sorted by the true target value in ascending order.*

due to the fact that the range of values of the generated data is of the same magnitude as the inaccuracies of the Forest. Still, the MLP manages to predict these values much better.

2 Training Process

Karl Thyssen

In reinforcement learning an agent is placed into an environment where it learns to interact with it dependent on the parameters a 'trainer' provides it. Ideally the trainer wishes the agent to embody certain desirable characteristics. This optimal policy is determined by the methodology the trainer uses to reward the agent. Rewards the trainer supplies the agent with are based on the state of the environment at the time the agent is to make a decision and the reward scheme weighting on this given state.

Initially the agent is often allowed to take random actions to influence the next game-

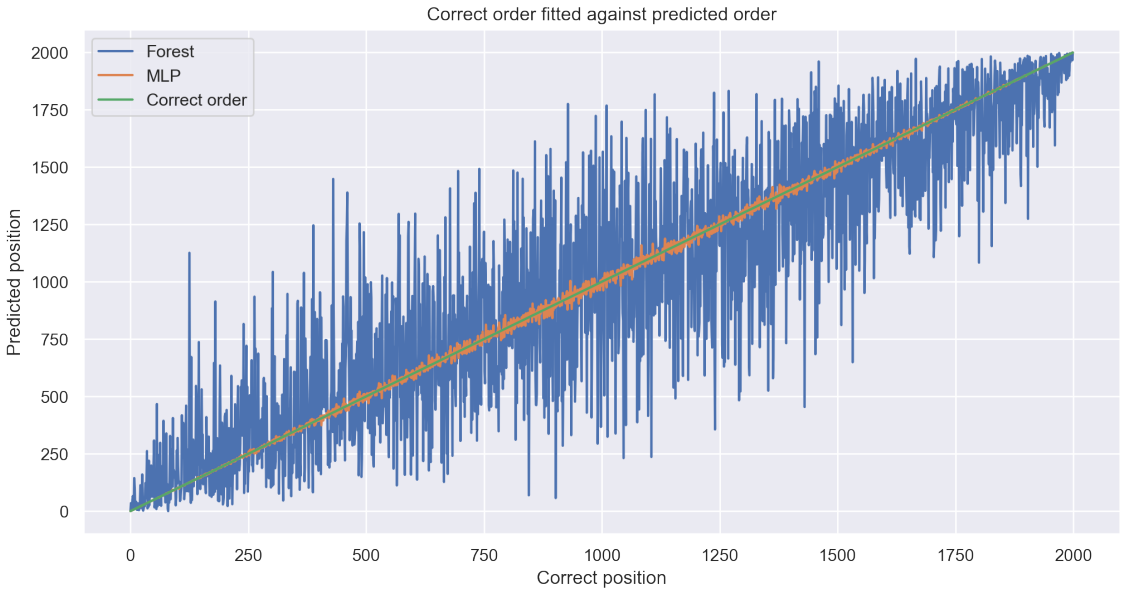

Figure 2: *This figure displays the same data as Figure 1. Here, the x-axis represents the position of each instance sorted by its target value, the y-axis displays the position if the instances had been sorted by the predicted target values of the respective regressor. The more the predictions resemble the correct order (the green line) the better the regressor is suited for our task.*

state. This random acting forms the agents initial policy π. The policy determines how the agent acts in the game, the ideal policy being the ideal way to play the game from the initial game-state.

The rewarding of an agent for a particular action a, functions to provide information as to the quality of the decision it made to take this particular action at this particular state s; this $\{(state, action)\}$ pair (s, a). Through this feedback the agent can update its policy. Through regular updates of the policy(π), after receiving a reward for games played based on the current policy the agent will begin to favour actions it predicts will lead to higher rewards hence improve the policy and bring it as closely as possible to the optimum.

Formally this is known as the Markov-Decision-Process where the state of the environment at a particular step t in the game is represented as s_t and the action taken for this time step t is represented as a_t. The current state is input into the current policy which leads the agent to select its next action. The action is selected from the action space $\mathcal{A}$ =$\{\text{UP}, \text{LEFT}, \text{DOWN}, \text{RIGHT}, \text{BOMB}, \text{WAIT}\}$. The reward for this action r_t is then given by the reward function $r(s_t, a_t)$ given which the agent

can then evaluate this move for the update of its policy. In Bomberman its simple to see that the state s_t is given by the distribution of agents, loot, bombs, coins etc. in the game map represented as the trainer wishes.

In Bomberman an action can not be rewarded entirely isolated from the game as a whole as actions have direct repercussions least 6 steps into the future (by the time the explosion has dissipated) and even beyond. Therefore the action must be rewarded based on all future steps with those very far in the future having decreasing impact. Therefore we introduce a discount factor $\gamma \in [0, 1]$. We arbitrarily selected 0.9 to ensure the steps $n+4, n+5, n+6$ have a non-negligible impact on the reward, as these are the steps that we hope to cash in from a bomb and the agent should see that this was a well placed bomb if this is the case. Unfortunately we see this being very negative as the agent often self-destructs...

Therefore the reward function that the agent is wishing to maximise the expected long term reward $E[\text{Reward}|\pi]$ for the optimal policy is:

$$\text{Reward} = \sum_{t=0}^{\infty} \gamma^t r_{t+1}$$

.

The concept of learning refers to the improvement of this policy, by two processes, policy improvement and policy evaluation. As the initial policy in our case will be created by either the random actions of the `random_agent` or the deterministic actions of the `simple_agent`, the initial policy evaluation step will take place on this gathered data. This is achieved using a value function for each of the states visited in this policy π. The value function V for each state s_t is the Expected value given the starting state is s of the discounted reward function of the $\pi(a_t|s_t)$ particular pair.

2.1 Q-Learning

Karl Thyssen

With our selected state vector of length 532 it would be highly impractical of not impossible to calculate the state action pairs for every state and the expected reward of each action due to the massive amount of data required for this along with the computationally intensive time requirement. Therefore we use the approach of Q-Learning to find the optimal Q function Q^* for each state-action pair in the data, updated using the discounted reward received for the action. We do this in the callback function of the agent when `end_of_episode()` is called, the discounted rewards are filled into the places of the previously stored immediate rewards of each state-action pair.

This Q function will need to be updated to find the optimal Q function by gathering data, evaluating the data and updating. To gather data we have 3 options: update Q after every step, after every n-steps or after every n-episodes. Due to the fact that we selected random forest regressors to store our Q the last option seems to suit the purpose best as it is not possible to update sklearn random forest regressors, they must rather be entirely trained from scratch. Therefore it would be incredibly time and storage inefficient to train them on multiple steps, particularly as they will not be effective with small data pools anyway. Therefore we opt to update Q after 10000 as suggested in [2].

We store our Q-function as 6 random forest regressors; one for each action, from which the agent will choose an action based on the stochastic policy of the trained forests, each supplying the expected reward for the action it is trained for. This action $a_{t+1} = \pi(a_t|s_t)$ will, in training, be mapped onto the calculated reward along with all state action pairs in the 10,000 games played during each particular policy iteration to retrain the forests, that is to say update the Q-function.

2.2 Exploration and exploitation

Hein-Erik Schnell, Karl Thyssen

While optimising the Q function based solely (exploitative approach) on the games of the previous generation (the trees trained on the past $10,000$ games) it is possible to slip into a local maximum in terms of the yield which may well not be the optimal Q. Therefore we introduced a degree of exploration using an ϵ-greedy policy in which a random action is performed with the probability $\epsilon \in [0, 1]$. We selected $\epsilon = 0.25$.

Max-Boltzman-exploration

When exploration occurs (not the best action is chosen), we decided not to choose the action completely random. Instead, choosing more promising actions should be more likely. This means that the expected rewards need to be mapped onto probabilities with which the respective action is chosen. The probability is given by

$$\pi(s, a) = \frac{e^{Q(s,a)/T}}{\sum_a e^{Q(s,a)/T}}, \tag{1}$$

where $\pi(s, a)$ is the probability that the action a is chosen given state s, $Q(s, a)$ is the expected reward if action a is performed after state s occured, $a \in \mathcal{A} \setminus a_{max}$ are all available actions after the action with the highest expected reward a_{max} has been deleted from the set of actions $\mathcal{A}$ because a_{max} would have been chosen if we didn't explore, and T is a temperature-parameter which scales the ratios between

the probabilities. We found that T should be of about the same magnitude as the expected rewards which is why we chose T as the mean of the absolute values of the expected rewards. Figure 3 shows an example of this. We later decided to enable

$Q(s,a)$	$\pi(s,a)$
43	0.52
21	0.25
4	0.14
-19	0.07
-65	0.02

Figure 3: *Exemplary table of expected rewards $Q(s,a)$ and corresponding probabilities $\pi(s,a)$ if T is chosen as described above. In this case:* $T = 30.4$

exploration only in one fourth of all rounds. The reason is that the agent can only survive after it placed a bomb if it moves consistently away from it or around a corner. Thus, a random move in every fourth step may be fatal. With the above measure, we could at least ensure the survival after placing a bomb and make the agent learn this behaviour.

2.3 Training data

Karl Thyssen

We generated new training data after each training generation of $10,000$ games to train the new forests that is to say the Q function for the next generation to learn from. Therefore the size of our final training data array of action state vectors was dependent on the number of rounds the agent survived in a given generation. We saved each set of training data independently of each other to allow for training from any generation and later access to the data. The random forest regressors were also saved to allow for analysis of them and comparison between the generations by allowing play from any generation and even play against younger generation. Ideally play against younger generations would always lead to free wins as the agent should be significantly better after some time, however our agent is a lonely agent that rarely sees others anywhere other than in its state vectors.

3 Results

3.1 Overview

Karl Thyssen

Here we will discuss the results we saw for the various approaches we attempted.

As previously mentioned we experimented with a γ value of 1.0 and 0.9 after which we continued other variations at $\gamma = 0.9$. Other variations include:

- $\gamma = 1.0$:
  - Generation 0 data generated from 4 *simple_agents*
    - Additionally train MLB
- $\gamma = 0.9$:
  - Generation 0 data generated from 4 *simple_agents*:
    - self-train until agents begin to regularly interact with each other
    - self-train
    - state vector reduced to 180 elements

3.2 The initial approach

Karl Thyssen

Having decided on the random forest regressors as the medium for storing our Q, and knowing the random forest has the ability to determine feature importances and weightings itself we decided to throw every feature we could think of at it as described in 1.1. We initially also chose a γ value of 1.0 for simplicity in testing and began to train an agent. In [2] the agent trains for 10000 episodes per generation and begins to show a large improvement between generations 0 and 20, however that agent is trained using a neural network so we expected a longer wait before seeing such spikes in performance. This is due to the erratic nature of the predictions made by the forests in relation to the MLP we tested.

Out of interest we also tested the same set-up using an MLP regressor, however unlike the forest, the impact of using any and all features was an exorbitant time required for the fitting after each generation, often longer than the data generation itself.

The time required to generate data for and fit each generation was between 4800 and 10000 seconds for the random forest. This time required is clearly dependent not only on hardware and thinking time but also the number of steps the agent takes per episode which was, unfortunately, consistently short.

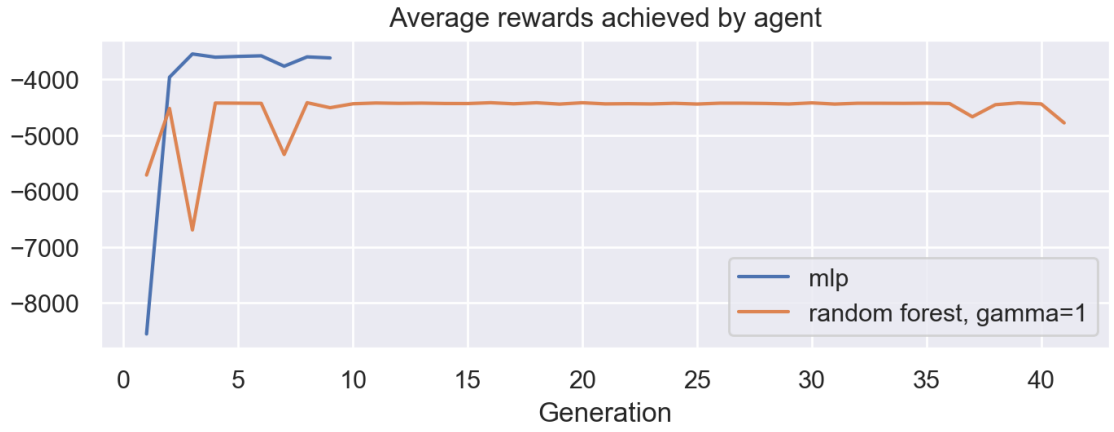

Figure 4: *Here we see the performance of 40 generations trained with the random forest and 8 generations using the MLP. We chose not to train the MLP further as the time constraints were too severe and we were not seeing any improvement.*

After these results we assessed potential reasons for such a static reward graph, that is to say a plateau in the rewards attained. To improve the quality of the next agent we looked at the following aspects of our learning process:

- Firstly we looked at the exploration policy which originally effectively lead the agent to choose a non-π based action every 4 moves with $\epsilon = 0.25$. We realised this would probabilistically cap the number of moves the agent would do before placing a bomb in a possibly precarious situation with $\#Action = 6$ at $\frac{6}{0.25} = 24$ moves. We chose to edit the exploration structure to follow what we discuss in 2.2, to ensure exploration effectively occurs in $\epsilon * 100\%$ of games rather than in every game, effectively increasing exploitation. Although exploitation leads to plateaus in general once the agent has fully exploited a particular policy it is using, we hope it explores often enough to prevent this and see more danger in the agent never learning how to play further than 24 moves.
- Secondly we looked at our rewards which originally offered 100 points for a $\{coin\_found\}$, 10 for an $\{invalid\_move\}$, 500 for a kill and -400 for a death (that is to say loosely following the scoring scheme). We chose to exaggerate the rare but important events(coins, kills, death) to ensure that *when* they

occurred, they were noticed. We also harshly punished the $\{invalid\_move\}$ as our first goal was to train an agent who knew how to manoeuvre through the map.

- Thirdly, after implementing a method to show the agents selected move, after a few generations we grew frustrated with the sheer quantity of $\{WAIT\}$ actions the agent selected, and increased the punishment for the action to -5. This seemed to have the desired effect as 'wait until I die' runs were not seen again.

3.3 Improving the learning process

Karl Thyssen

Altering γ

After seeing the poor results in terms of learning improvements of the random forest with $\gamma = 1.0$ we proceeded to introduce $\gamma = 0.9$. The hope was to allow the agent to differentiate between a good move and a poor move rather than between a good game and a bad game by isolating the individual moves more from later ones. However due to the delayed effect of the bombs and explosions the move itself **must** include rewards from later moves to some extent. While keeping all else equal the change in gamma value resulted in the agent surviving for noticeably longer than previously, the overall reward however doesn't seem to have benefited in the same way. This is despite an increase in the agents score, we assume this is due to the increased moves, costing the agent the points it receives for randomly picking up coins.

This indicates that some learning takes place as the agent no longer commits suicide early. Other points to support this are that most actions the agent performs must be valid, hence the discrepancy to the $\gamma = 1.0$ agent's baseline of only 200 points. Therefore the agent must have learnt how to make valid moves in the initial starting positions and must sometimes collect rewards(hence the points). This is corroborated purely qualitatively as we printed the agent's moves' validity and could observe a decrease in invalid moves after the first two generations. Unfortunately there doesn't seem to be a positive trend in any on these metrics to indicate improvement other than potentially a very small gain in survival steps.

streamlining the state vector

Our initial approach having been the 'the more the merrier' variant we decided to assess look at the state vector again to remove redundancies. There are many alternative methods to present the state vector, which we will discuss later, wishing to stay close to our original schematic we selected however to simplify our original

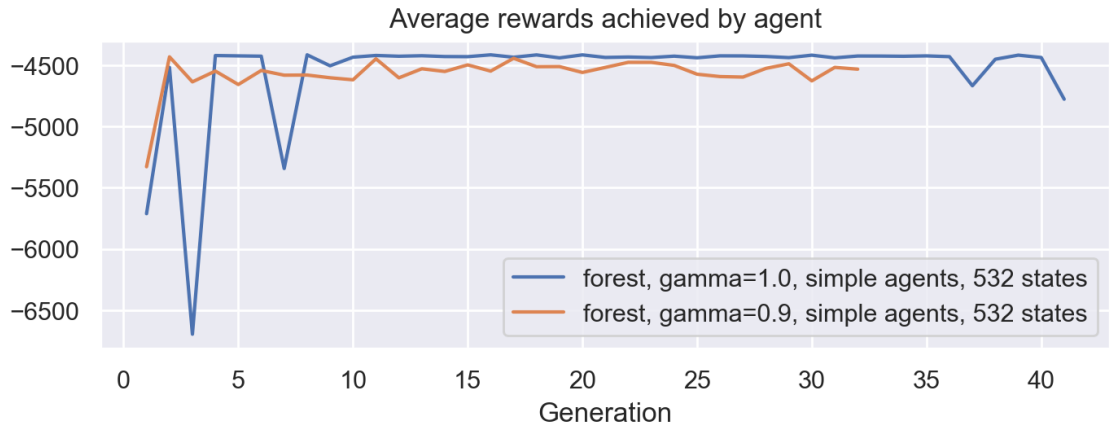

Figure 5: *Here we see the performance of the random forests trained with their respective gamma values, clearly there is no great difference in the rewards other, they both plateau within around 2 invalid move values of rewards of each-other. The relevant change is noticed in the rewards and survival graphs that follow.*

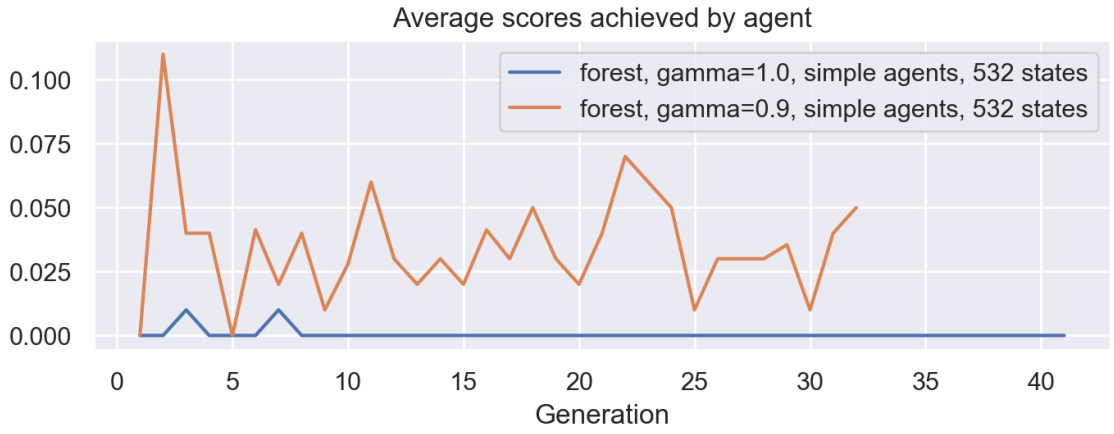

Figure 6: *Clearly the scores for the agent are not consistent throughout the generations, the primary difference being the fact that in 100 games the* $\gamma = 0.9$ *agent consistently collects the equivalent of 3 coins.*

rather than changing it conceptually. Therefore we implemented the modified state vector in 1.1.

As we can see, the new agent does not achieve a greater average reward while consistently surviving longer and achieving a higher score. Overall a longer life while maintaining a close to equal reward implies a higher quality of moves as the negative

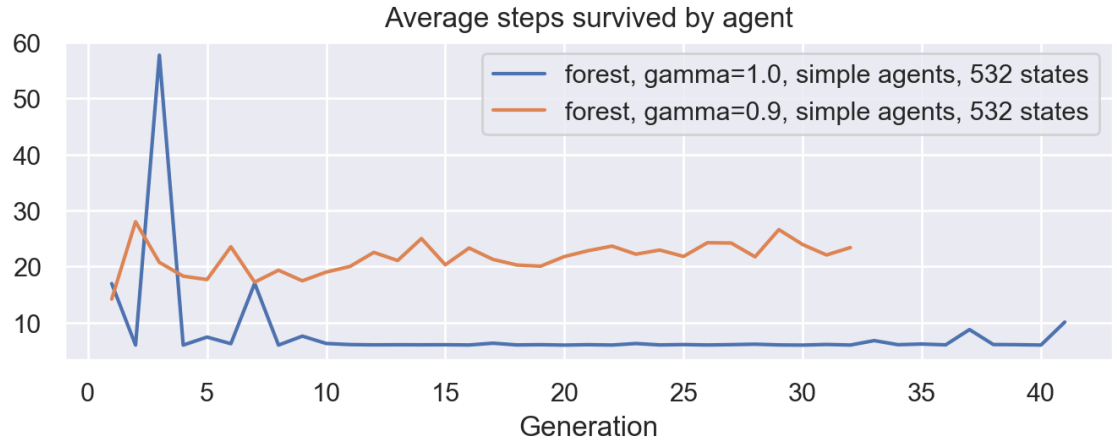

Figure 7: *While the agent with $\gamma = 1.0$ plateaus at 6 steps of survival, indicating his first action is to place a bomb and die to it after 5 steps, the other agent survives around 20 steps longer.*

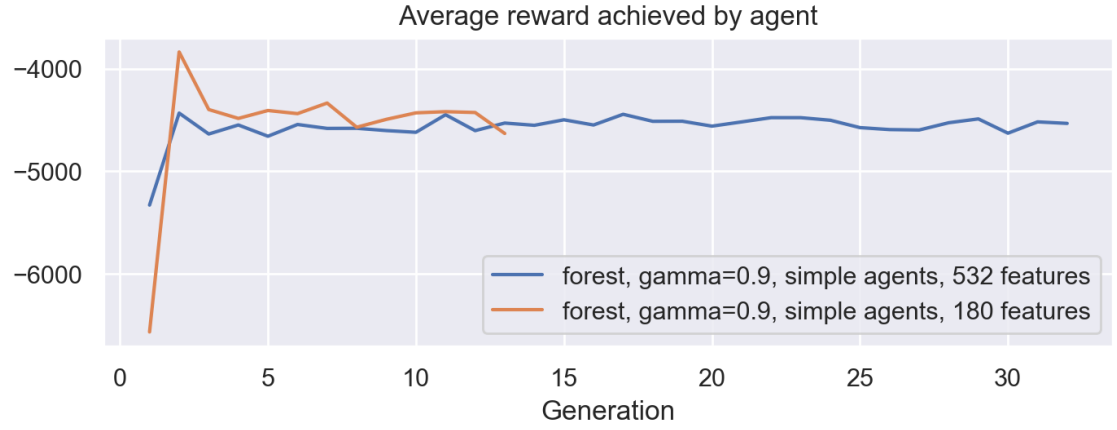

Figure 8: *As before, the reward graph is disappointing as it seems the agent does not learn to increase his reward any further despite clearly (as can be seen from the scores) not playing the game optimally.*

reward obtained from moving at all, particularly if there are any invalid moves among this (as they are harshly punished) must be balanced out by for example destroying crates or obtaining coins.

At this point the next logical step, in hindsight, would have been an overhaul of the rewards, perhaps in some of the ways we explore later. However due to our earlier changes to the rewards and the fact we did not know how best to test for the best

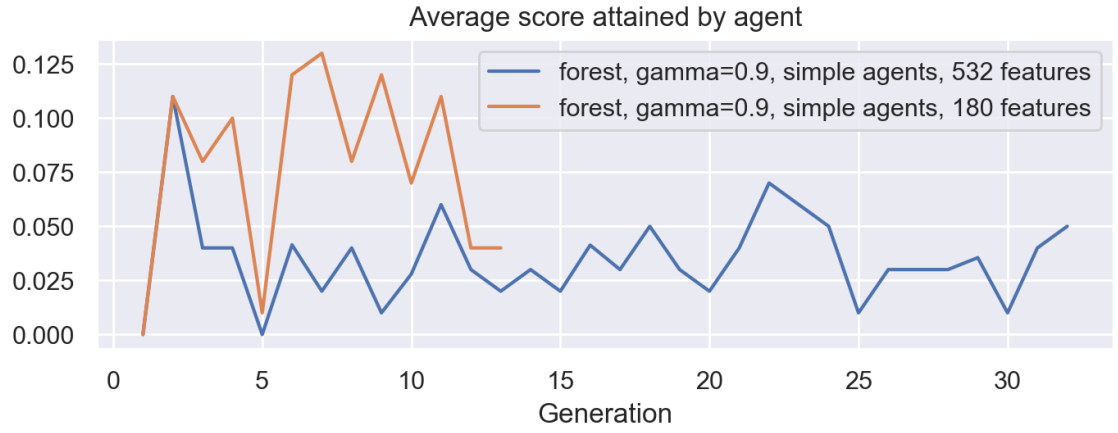

Figure 9: *The scores achieved for the agents trained in identical conditions other than their feature representation. Clearly the prevailing trend is for the scores of the simplified agent to be greater than those of the* 532 *feature agent.*

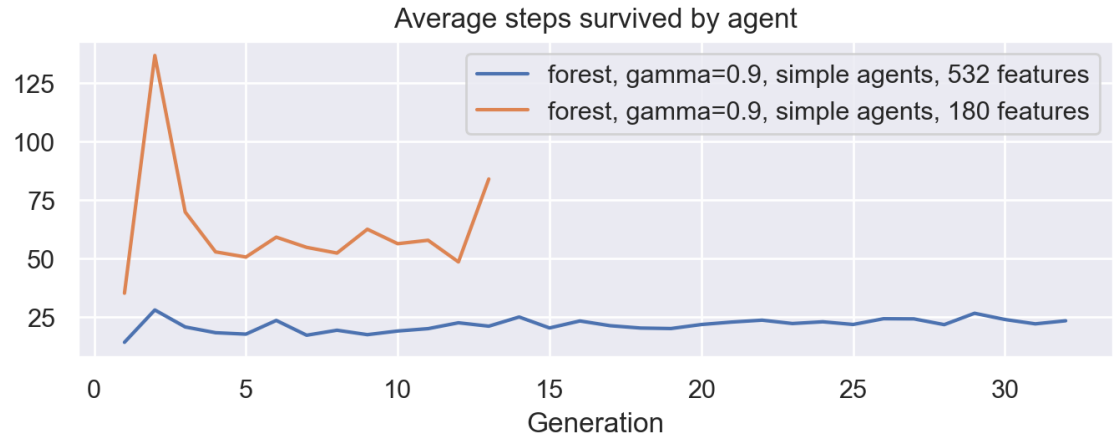

Figure 10: *The agent with its state vector consisting of* 180 *features consistently survives longer than the one with the lager state vector.*

reward scheme we did not try variations on the rewards.

Training changes

We had a suspicion that as our agent does not interact with the simple agents of its own accord, that is to say it stays isolated in its own corner, the fastest way to train it would be to self train it with 4 of itself in the arena until it gets to a point when it starts to meet its counterparts. However from our tests of this it seems that

although it survives for a lot longer than all other agents it still does not seem to learn how to move out of its box to interact with other agents, merely it happily walks left and right until it randomly drops a bomb or reaches 400 steps.

Final Data

Finally the spectrum of agents we trained makes for a sorry sight in the graphs:

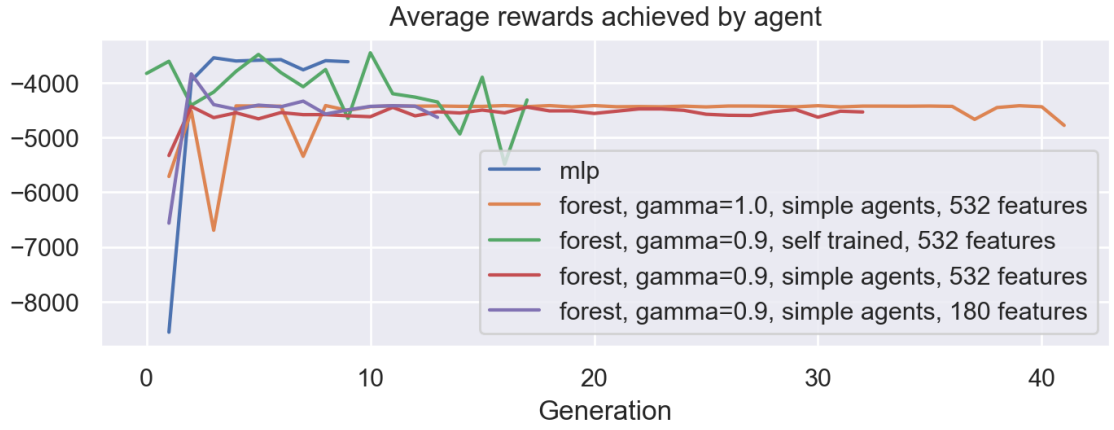

Figure 11: *Shows the average reward earned by each agent in each generation.*

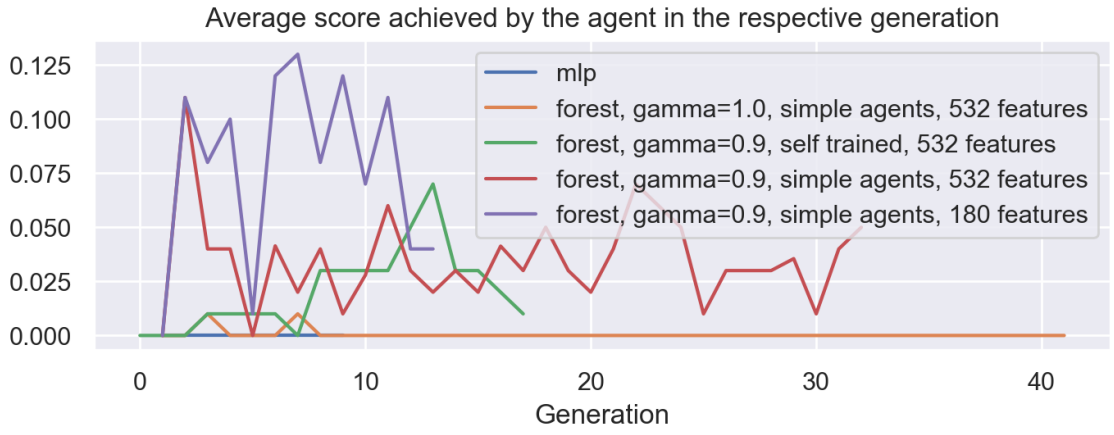

Figure 12: *Shows the average score attained by each agent in each generation.*

We will submit the agent using 180 features in its state vector with $\gamma = 0.9$ as this seems to be the agent most consistently getting the highest score over the generations, at a maximum of one coin per ten episodes.

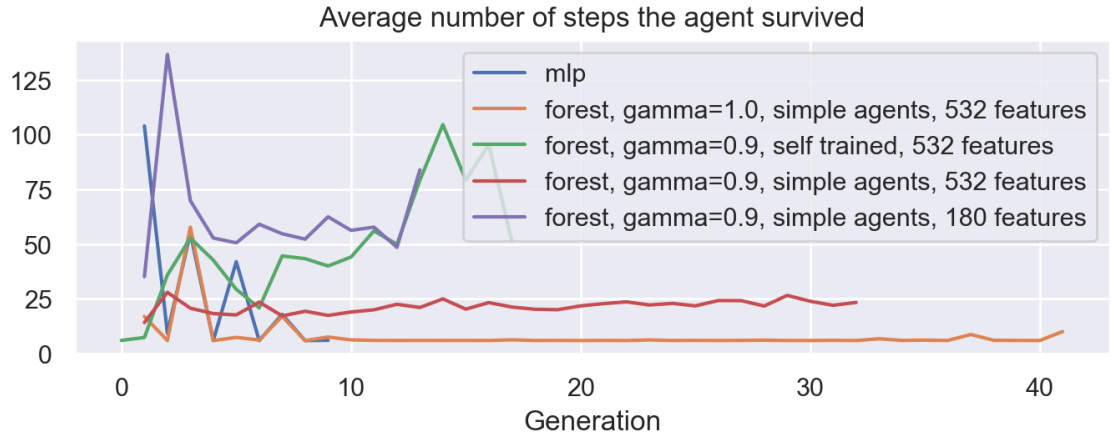

Figure 13: *Shows the average number of steps by each agent in each generation, shows how self training lead to increasing survival with negatively trending reward.*

3.4 Outlook

Hein-Erik Schnell, Karl Thyssen

As discussed in the previous subsection, our agents are very unsuccessful. They mostly remain in their starting corner and more often than not they blow themselves up. It is therefore necessary to discuss alternative approaches to the task and how we could have improved our implemented approaches.

As we see it, the topics to be discussed are mostly the sections above:

- State representation
- Reward scheme
- Regressors
- Training procedure
- Exploration / Exploitation

State Representation

With what we saw so far, we have to assume that less features in the state vector lead to far less possible states and better performance of the agent. The 180-features version of the state vector is already close to the smallest form of state vector possible if we want to represent the whole game board, which would be at least 176 features (176 accessible cells).

A possible improvement would be a state vector which does not represent the whole game board. The simple agents, for example, only know the position of possible targets and whether they are in danger. With this in mind, we could create a state vector which gives only the informations relevant to the simple agents. Given that in most cases most of the game board is not relevant (especially at the beginning of an episode), this seems to be a quite reasonable approach. It would also help the agent to prioritize certain features, e.g. avoid explosions before collecting coins.

Reward Scheme

In the plots above, we see that achieved scores and rewards hardly correlate. In the ideal case, higher score means more rewards. Since this is not the case, we need to revisit our reward scheme and distribute much higher rewards for what actually scores. On the other hand, our agent died a lot which meant lots of penalties. But we don't get negative scores for dying. We don't really know which way is better and we would have tried both ways in order to find out.

Regressors

The regressors were the third crucial design choice. As hinted above, choosing a regressor which is very erratic (Random Forest Regressor) and one which we don't know much about (MLP-Regressor) left us somewhat dissatisfied. Still, they were the most practical choices to us.

For both regressors we used almost the default settings. Adjusting those settings accordingly to our task might have helped a lot. But in the case of the MLP-Regressor, we could only guess what effects the various parameters might have.

All in all, choosing a better suited regressor and adjusting its parameters properly are two of the most important tasks in order to improve our model.

One should also mention that, after each generation, we fitted a whole new Forest. One could try to adjust the former Forest towards the data, instead.

Training procedure

We mostly trained our model with 10000 episodes per generation. In one case even for more than 40 generations (i.e. 400000 episodes). Since Reinforcement Learning sometimes requires long times before encountering a good policy, we could never know whether our agent is not able to learn properly (because of the points above) or if it just didn't have enough time. With almost no experience from prior projects or exercises concerning RL, we didn't know how to find out which is the case. Speaking

of only the training procedure, having more episodes per generation and letting it run for more generations might always improve the results.

Exploration / Exploitation

As mentioned before, the agent tends to kill itself if *exploration* is enabled because it is likely to take random actions shortly after having placed a bomb. In order to prevent this, one could lower the ϵ significantly. Apart from that, we think that *exploration/exploitation* is of less importance in terms of improving the model. The bottleneck are the points mentioned in *State Representation, Reward Scheme* and *Regressors*.

3.5 Further Comments on the final project

Hein

: As already implied in several parts of the report, our greatest issue was the lack of knowledge and experience with Reinforcement Learning. Some exercises with feedback and sample solutions would have helped a lot. For the next time, I propose to divide the final project into subtasks which are then given to the students as the final one or two exercise sheets and let the final project be something like the final improvements to an already working environment and model.

Karl

: I think it would have been very helpful/interesting to learn before the project or as part of the project itself how to adjust the parameters that are used to make them feel less arbitrary. A large part of the problem was that we had no idea whether an adjustment would be effective and how or where to adjust parameters to have the desired effect as training to test for an impact would take days. Overall I did thoroughly enjoy the project but would have appreciated some additional guidance to ensure I also learnt how it is done properly.

Bibliography

[1] Richard S. Sutton, Andrew G. Barto (2018)
Reinforcement Learning: An Introduction
Available at: http://incompleteideas.net/book/bookdraft2018jan1.pdf

[2] Joseph Groot Kormelink, Madalina M. Drugan and Marco A. Wiering (ICAART 2018)
Exploration Methods for Connectionist Q-Learning in Bomberman
Available at: https://bit.ly/2GReSxQ