Fundamentals of Machine Learning

Final Project

Applying Reinforcement Learning to Bomberman

Submitted By :
Hein-Erik Schnell
Karl Thyssen

Contents

1 Learning Method and Regression Model 2
1.1 State representation 2
1.2 Rewards 4
1.3 Regressors 5
2 Training Process 6
2.1 Q-Learning 8
2.2 Exploration and exploitation 9
2.3 Training data 9
3 Results 10
3.1 The initial approach 10
4 Heins thoughts 11
4.1 State representation 11
4.2 Rewards 13
4.3 Estimator 13
4.4 Learning algorithm 14
4.5 Necessary functions 14
5 Karls thoughts 15
5.1 Neural Networks 15
5.2 Classical approach 16
6 Karls observations 18
6.1 The state matrix 18

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 enemy** $\{0, 1\}$:
    `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.

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**: $+100$

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.

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 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 weighting of the 'trainer' on certain desired characteristics. This optimal policy is determined by the rewards the trainer supplies the agent with, which are based on the state of the environment at the time the agent is to make a decision. 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 $\mathbf{A} = \{\text{UP}, \text{LEFT}, \text{DOWN}, \text{RIGHT}, \text{BOMB}, \text{WAIT}\}$. The reward for

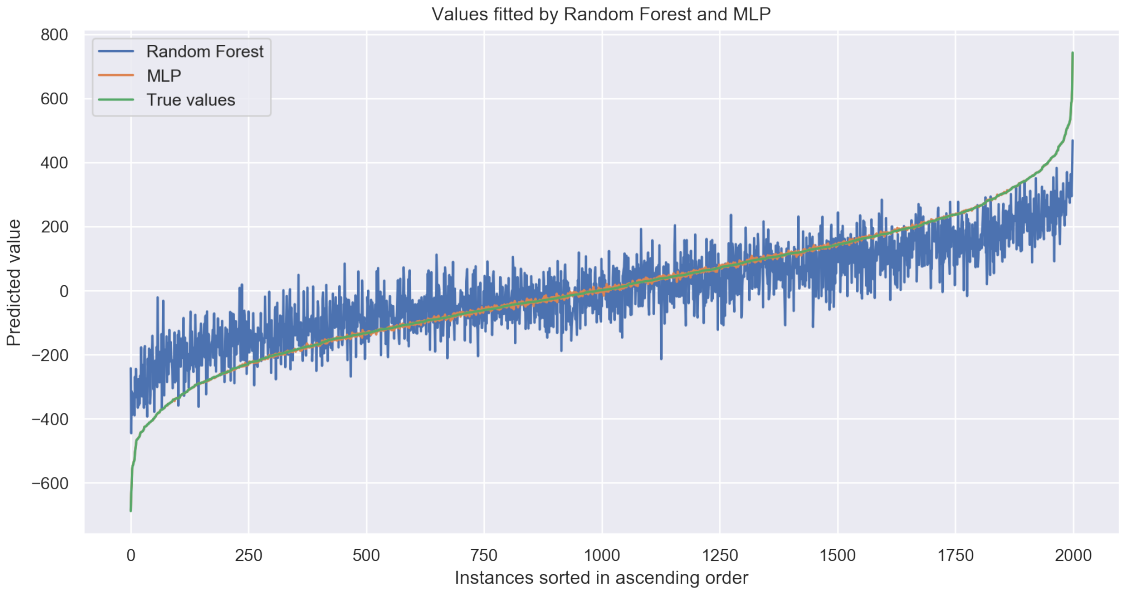

Figure 1: For this figure, `sklearn.datasets.make_regression` was used to create 10000 samples with 100 features of which 10 are informative. We then trained the Random Forest Regressor and the MLP-Regressor provided by *scikit-learn* with 80% of the generated data. The figure shows the predicted values 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.

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

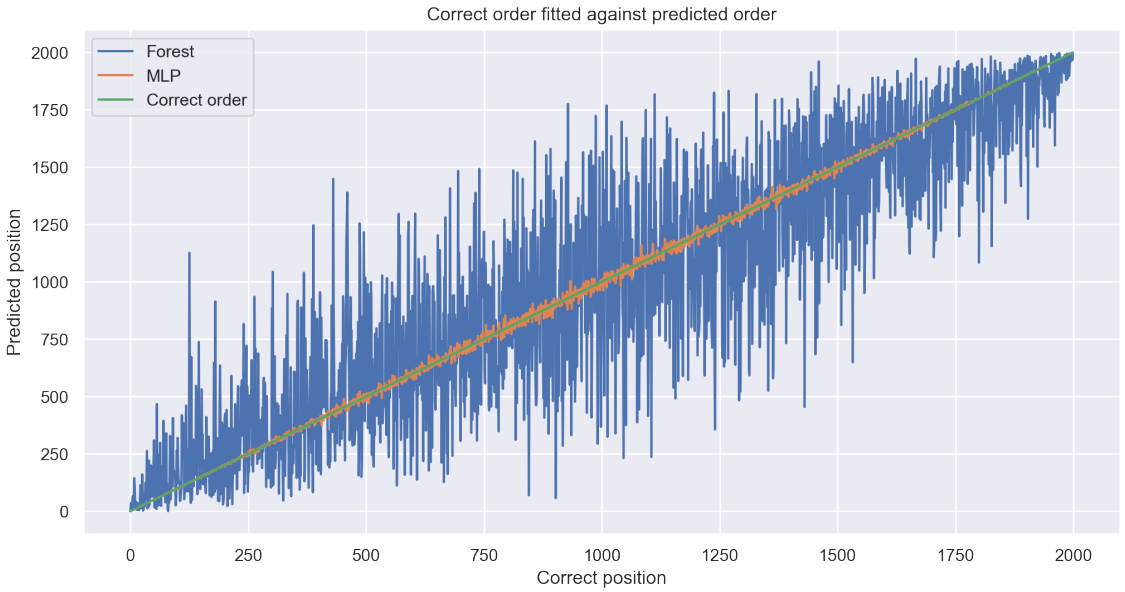

Figure 2: This figure displays the same data as Figure 1. Here, the x-axis represents the position of each instance sorted by its target value, the y-axis displays the position if the instances had been sorted by the predicted target values of the respective regressor. The more the predictions resemble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

long term reward $E[\text{Reward}|\pi]$ for the optimal policy is:

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

.

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 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.

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

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 introduce a degree of exploration using an epsilon greedy policy in which we introduce a random action with the probability of $\epsilon \in [0, 1]$. We select $\epsilon = 0.75$. Simply put, while learning, the policy selected for a particular state action pair is

$$\pi(a_t|s_t) = \begin{cases} argmax_{a \in A} Q(s_t, a) & \text{if } x \geq \epsilon \\ a \in A & \text{if } x < \epsilon \end{cases}$$

with $x \in [0, 1]$ selected at random at the time of the action decision being made.

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

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
  - Generation 0 data generated from 4 *random_agents*:
    - self-train until agents begin to regularly interact with each other
    - self-train

3.1 The initial approach

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 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 10 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 average time required to generate data for and fit each generation was roughly 4800*seconds*. 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.

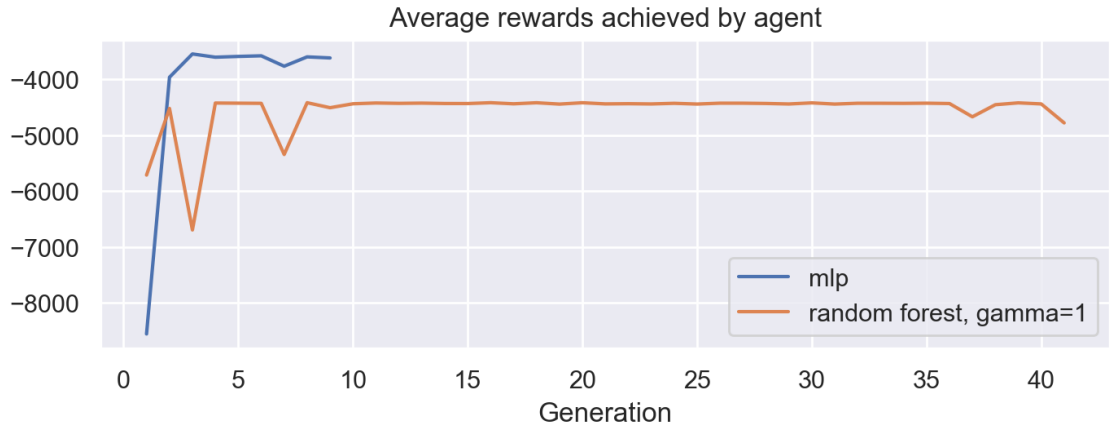

Figure 3: Here we see the performance of 40 generations trained with the random forest and 8 generations using the MLP. We chose not to train the MLP further as the time constraints were too severe and we were not seeing any improvement.

4 Heins thoughts

Tasks to be done are:

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

4.1 State representation

My proposal is a 2D numpy array. Each row (first index) represents a single state. Each column represents a feature. All available information is stored within each agent in the dictionary `self.game_state`. What features do we store?

- For each cell:

  - **Crate, wall, free** $\{1, -1, 0\}$: The same representation is provided by `self.game_state`. Only it has to be reshaped into a 1D-array.
  - **Cell contains agent** $\{0, 1\}$
  - **Cell contains opponent** $\{0, 1\}$: This and the point above need to be stored seperately because it is possible for agent and opponent to occupy the same cell. It would just be impossible to distinguish whether a cell is occupied by just one or more opponents. But this should be a negligible issue.
  - **Cell contains coin** $\{0, 1\}$
  - **Explosion on cell**: Provided as 2D-numpy array by `self.game_state`.
  - **Danger level** $\{0, \ldots, 4\}$: How many time steps until an explosion will hit the cell.

- Just once at the end of the array:
  - **Current step number** $\{1, \ldots, 400\}$
  - **Bomb action possible** $\{0, 1\}$
  - **Danger level:** This is a danger level for the agent. It is not necessary but provides a clearer measure of whether the agent is actually in danger. I propose to calculate this by multiplicating the danger level of each cell with the *Cell contains agent* entry of each cell and then take the maximum of all the results. This way we will only get a non-zero value if the agent is on a cell with a danger level > 0.
  - **Reward already received in this episode**
  - **Reward received at the end of the episode**: This one has to be added subsequently at the end of the episode to each ocurred state.
  - **Reward gained after this state occurred:** This one is the difference of the two above. This should be our *target* which we aim to maximize.
  - **Agent touches opponent** $\{0, 1\}$: The simple agents consider dropping a bomb if they touch an opponent. This may be a very good indicator for dropping a bomb for our model.

This gives us 6 entries for each cell and 7 entries at the end of the array. There are 17×17 cells in the arena. However, the cells located at the rim are always *walls*. These do not need to be stored. What remains is a 15×15 grid. I am not sure whether we want to store cells with *walls* at all, because these are always the same.

They should add nothing to the state of the game.
Therefore we have $15 \times 15 = 225$ entries ($15 \times 15 - 7 \times 7 = 176$ if we don't store walls) plus 7 at the end of the array for each state (time step). Storing these total 232(183) entries in a numpy array might produce big amounts of data. We should consider specifying the data type explicitly. However, the numpy documentation states that the data type is chosen as the minimum type required to hold the objects in the sequence.

4.2 Rewards

The most rewarded actions should be those which will win us the game. Those are collecting coins (1 point) and killing opponents (5 points). Therefore, I propose the same scaling between those two when rewarding them.
Subgoals which we consider helpful for winning should be rewarded too, but only with few points. It should be almost impossible to substitute the *winning actions* (coins and killing) with *helpful actions* (destroying crates).
I also propose a penalty of -1 for every action so that the agent learns to act as efficiently as possible.
Dying should be (beware! wordplay:) gravely punished.

Action	Reward
Collect coin	100
Kill opponent	500
Destroy crate	1-2 per crate
Perform action	-1
Die	-500fd

4.3 Estimator

I don't know much about neural networks which is why I spent most of my thoughts on how to implement this with the models we used in the exercises. Plus, I like the idea of the challenge to come up with an agent which doesn't use a neural network. Since we want to estimate the expected reward of performing an action at in a given state, we need a regressor. This regressor needs to be highly flexible in order to cope with the variety of states. I guess the most flexible regressor we used would be a *Random Forest Regressor*. On the one hand, this one would need tons of learning data but on the other hand, which proper regressor doesn't?
This means, we would plug the data of the states and the received rewards into the model and get an estimate of what reward we could expect for what action. Since we need to distinguish between the six possible actions, we need six Random

Forests. One for each action. Each forest trained only with the states after which the respective action was performed. This also means that we need to store all data in six different set for six possible actions or store the action performed afterwards in the *state vector*.

4.4 Learning algorithm

I propose to use the Max-Boltzmann (MB) method as described in [2] or something similar. It is mostly a ϵ-greedy algorithm. But when it decides to explore, it doesn't choose randomly among the remaining actions but assigns probabilities corresponding to the value (expected reward) of the remaining action. This way, exploration is not just random but more targeted.

There is no point in letting the agent learn after every single episode. In the long run those learning interruptions will cost us a lot of time. Plus, it is very unlikely that the next episodes are similar to the one before which means that the learning will most likely not be applied in the subsequent episodes. Thus, the learning process itself should be divided into learning intervals of many episodes, say 100 to 1000, maybe even 10000 episodes, given the number of possible states. While we're already using the term *episodes*, let's call those learning intervals *seasons*.

I propose the following procedure:

1. Use the provided simple agent as our agent to play the first season. I would use the MB algorithm already in the first episode. This already gives us the possibility of exploration which I consider as important since the simple agent is fully determined and hard coded.
2. Use the data of the first season to train our model.
3. From now on, let the agent rely only on the predictions of our model. Create data, explore according to the MB algorithm.
4. Use the data of all former seasons to again train our model.
5. Repeat the two former steps.

The simple agents should be good sparring partners to start with. They are very good at avoiding bombs but sometimes lack the ability to collect coins. We might improve this ability in the simple agent code to get better starting conditions. Maybe by automatically decreasing the distance (and thus increasing importance) of coins.

4.5 Necessary functions

Some functions that need to be implemented in order to prepare everything:

- Calculate danger levels of each cell
- Insert *coins* into *state vector*
- Insert *opponents* into *state vector*
- Insert *self* into *state vector*

We need to decide what structure the *state vector* should have. There are two ways:

1. All features of a concerning cell (coin, opponent, self, danger level,...) subsequently. The all features of the next cell and so on ...
2. All data of one feature (e.g. coin) for all cells, then all data of the next feature for all cells and so on ...

In both cases, the functions that insert the values into the array should make use of the slicing possibilities of numpy arrays (e.g. `x[starting point:end point:stepsize]`). In the first case, only every sixth entry represents the same feature. In the second case, the first hundred entries represent only one feature for different cells.

5 Karls thoughts

How this project can be approached:

- Neural Network approach *vs.* Classical Machine learning approach
  - Supervised learning *vs.* Reinforcement learning *vs.* Unsupervised learning
  - Feature selection to determine relevant information the agent should "see" and learn from
- Data management for training - not necessary as we will be collecting all the relevant data while training rather than using a large pre-compiled dataset

As we do not have access to more than our own laptops which can not be classed as high performance systems with low end dedicated GPUs training neural networks may be impractical and time consuming.

5.1 Neural Networks

As we didn't cover neural networks in the lecture but are allowed to apply them to this Project I explored it as an option despite knowing we are likely to select the classical approach. Some algorithms that are worth exploring are:

- Supervised learning:
  - Gradient descent involves forming a continuous error function using rewards to minimize
- Reinforcement learning:
  - Q-Learning (as covered in lectures)
  - Genetic Algorithm:
    * Training occurs through improvements from generation to generation by improving the strength and bias values for the edges of the neural network (stored in a 1D array). Each generation has a set number of instances that follow a variation of the previous generations values. To decide which of the instances in the previous generation had the best strength and bias values a scoring system must be created based on the performance of each instance.
    * Outputs are predetermined as 5 movement options (up, down, left, right, stand still) and the Bomb placement option
    * Factors to be decided on:
      · Inputs e.g. *weighted score of distance to enemies/crates/coins/bombs*
      · Number of hidden layers and nodes (not sure how this is done yet)
      · Method for genetic variation e.g. *ranking by score and weighting the likelihood for this process to be selected, random selection.* This is the exploitation aspect. Some values however should be randomly changed or multiple arrays crossed together (like nature) to increase the likelihood of finding the global minimum rather than local (exploration.

5.2 Classical approach

We could use Q-learning to train a ϵ-greedy agent. We want to find the optimal policy $\mathbf{Q}$ to solve the game.

Here its important to identify the 2 sets of attributes that contribute to decision making, **state** and **action**. The state will be input to decision making function to determine the action, the features that describe the state however must first be decided on. To reduce runtime it may be advantageous to perform a dimension reduction if it becomes clear that some features for example the states of the cells

outside of a 5x5 radius around the agent have very little impact on the decision. This could be particularly important in the early tests before the final train to reduce training time required when for example testing various reward value boundaries.

State appraisal

Before a decision can be reached as to the next action clearly the state of the board must first be assessed. With an almost infinite number of possible inputs possible it is important to have as few and as relevant features as possible.

Ideas to test/discuss:

- Are all accessible fields relevant
- Is it possible to approximate the state with knowledge about only the non-empty fields. In extreme cases this can be 2 (final 2 agents) or 176 (all fields occupied, although this is a stalemate) so this is likely not constructive.
- How should the distances to other agents or items be stored? Possibilities include:
  - The route that could be taken
  - Only the coordinates
  - The number of moves required to move the the desired opponent or coins field and the direction. Here the shorter route could also be through boxes and therefore include bomb wait times
- Should any previous states have a weighted input? - probably not, could be useful in some edge cases maybe?

Rewards

Naively we want to positively reinforce proactive, safe actions and negatively reinforce hazardous or even fatal actions so we should set our rewards based on this structure. This may however not always lead to the optimal solution as there are some situations where a sacrifice can lead to a greater boon in later steps e.g. *queen sacrifice for checkmate in chess.*

We have essentially 3 choices for how to issue rewards:

- Updating Q after each episode:
  - This leads to faster training times as the entire episode can play out without interruptions, however each episode leads to slower improvement

as the Agent can only perceive the results of all decisions made during the episode.

- Stepwise Q updates:
  - Very slow episodes (opposite of episodic updates above)
  - Ultimately it will depend on how we set our rewards, 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.
  - This would allow for punishment for 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
  - Decisions that are not allowed in certain situations such as movement into a wall/crate should also be punished harshly
- N-step Q updates: see above every N steps

Importantly the values for rewards should be small to ensure numerical stability in later operations

Exploration vs. Exploitation

- e-greedy policy

  [[learn to Tex maths]]

- soft max policy to max the "Temperature" S over time

  [[learn to Tex maths]]

6 Karls observations

These are just notes to document stages of development and note questions that arise.

6.1 The state matrix

- How should a bombs presence be documented in the 'state' matrix?
- Should a bomb timer be the input?

- Is there any value in including the fields that have an explosion in them? I can't think of a situation in which this would be relevant to making a decision for the next action.

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