Ruprecht Karl University of Heidelberg

Final project – Machine Learning essentials

„Reinforcement Learning for Bomberman“

from

Constantin Zehender and Jiufeng Li

Prof. Dr. Ullrich Köthe

Summer-Semester 2023

Premise

The following report explains the journey and development that finally led to the submission of the agent „train_rule_agent“. All the below-described agents were developed under the premise of using reinforcement learning strategies for the arcade game „Bomberman“.

The report will start with the very foundations of our analysis of the game and possible strategies, and end with the assessment of all developed agents under performance aspects in the final game settings.

Contents

1. Code availability statement

The framework for the game „Bomberman“ was cloned from the provided Git repository under the directory (https://github.com/ukoethe/bomberman_rl) and remained, if not mentioned otherwise, unchanged throughout the development of our agents.

The final code for our agents, as well as mistakes and older models, can be found at our Git repository under the address https://github.com/CZehender/ML_essentials_final_project_2023.

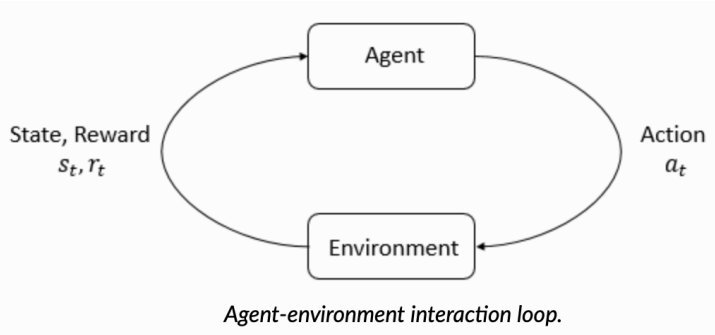
To be able to run all agents, the packages NumPy, PyTorch and keyboard need to be installed.

As highlighting the author of subsections is required, the author is given in brackets behind the title of said subsections.

2. Introduction

2.1 Overviews (Jiufeng. Li)

Reinforcement learning (RL) discusses the problem of how an agent can maximize the rewards it can obtain in a complex and uncertain environment. Reinforcement learning consists of two parts: the agent and the environment. During the reinforcement learning process, the agent interacts with the environment all the time. When an agent acquires a state in the environment, it uses that state to output an action, which is also called a decision. This action is then executed in the environment, which outputs the next state and the reward for the current action based on the action taken by the agent. The goal of the agent is to get as many rewards as possible from the environment.



Agent-environment interaction loop.

The main characters of RL are the agent and the environment. The environment is the world that the agent lives in and interacts with. At every step of interaction, the agent sees a (possibly partial) observation of the state of the world, and then decides on an action to take. The environment changes when the agent acts on it, but may also change on its own.

The agent also perceives a reward signal from the environment, a number that tells it how good or bad the current world state is. The goal of the agent is to maximize its cumulative reward, called return. Reinforcement learning methods are ways that the agent can learn behaviors to achieve its goal.

2.2 Problems (Jiufeng. Li)

The Bomberman game, known for its strategic gameplay and maze-like environment, presents an ideal testbed for evaluating the capabilities of RL agents. In Bomberman, an agent navigates a grid-based world filled with obstacles, crates, coins, bombs, and opponents. The primary objective is to

eliminate opponents and collect coins, while avoiding bombs and explosions. Success in Bomberman hinges on a combination of strategic planning, spatial reasoning, and adaptive decision-making, making it a challenging problem for RL.

Another thing is how do we construct good feature vectors, which is important for us to train the model at a later stage. The extraction of good feature vectors helps us to reduce the dimensionality of the feature vectors, thus avoiding the introduction of unnecessary computational complexity, and it is also more favorable for us to adjust the hyperparameters.

2.3 Overview of Available Methods (Jiufeng. Li)

1. Q-Learning:

Q-learning is a foundational algorithm in Reinforcement Learning. It operates on the principle of estimating the quality (Q-value) of taking a particular action in a specific state. The Q-value represents the expected cumulative reward an agent can obtain by following a given action in a given state and thereafter following an optimal policy. Q-learning uses an iterative update rule, often referred to as the Bellman equation, to refine its Q-value estimates over time.

2. Deep Q-Networks (DQN):

DQN builds upon Q-learning by incorporating deep neural networks. Instead of using traditional Q-tables to store Q-values for each state-action pair, DQN employs a neural network to approximate Q-values, making it suitable for high-dimensional state spaces like images. DQN leverages experience replay and target networks to stabilize training and improve sample efficiency.

3. Policy Gradients:

Policy Gradient methods take a different approach by directly learning a policy, i.e., a mapping from states to actions, without explicitly estimating value functions. These methods aim to maximize the expected cumulative reward by optimizing the policy parameters. Policy Gradients have the advantage of handling stochastic environments effectively and naturally accommodating both discrete and continuous action spaces.

4. Proximal Policy Optimization (PPO):

PPO is a state-of-the-art policy optimization algorithm. It seeks to address issues of stability and sample efficiency in policy gradient methods. PPO constrains the policy updates to prevent significant policy changes between iterations, ensuring that learning remains stable.

5. Actor-Critic Methods:

Actor-Critic methods combine the strengths of both value-based (like Q-learning) and policy-based methods. They consist of two components: the actor (policy) and the critic (value function). The actor suggests actions, while the critic evaluates those actions. This combination enables efficient learning by utilizing both the policy's exploration capabilities and the value function's guidance.

6. Deep Deterministic Policy Gradients (DDPG):

DDPG is an actor-critic method designed for continuous action spaces. It extends the DQN architecture to handle continuous action spaces and is particularly useful when applied to robotic control and similar domains. DDPG employs experience replay and target networks, similar to DQN.

7. Monte Carlo Methods:

Monte Carlo methods estimate value functions and policies through the averaging of returns from sample episodes. These methods are model-free and sample trajectories to learn about the environment. Monte Carlo Tree Search (MCTS) is an example often used in games that involve sequential decision-making.

8. Multi-Agent RL:

In the context of Bomberman, which often involves multiple agents, Multi-Agent Reinforcement Learning (MARL) methods can be applied. These methods consider interactions and dependencies between agents, allowing for more realistic training environments.

9. Distributed RL:

Distributed RL leverages multiple agents or parallel environments to collect experiences concurrently. By doing so, it accelerates training and enables efficient exploration in complex domains like Bomberman

3. Methods

3.1 Q-learning (Jiufeng. Li)

1. Algorithm Overview:

Q-Learning is a model-free, off-policy reinforcement learning algorithm used for estimating the optimal action-value function (Q-function) of an agent in an environment. In the context of the Bomberman game, the Q-function estimates the expected cumulative rewards for taking actions in different states (positions on the game board).

2. Design Process:

State Representation: game state, such as the positions of crates, coins, opponents, and the agent itself. The state representation facilitates us to perform feature extraction on the current Bomberman board so as to update the reward list according to different states.

Action Space: Q-Learning naturally suits discrete action spaces. In Bomberman, this includes actions moving up, down, left, right, bomb, and wait.

Exploration vs. Exploitation: A critical choice is the exploration strategy. We use ε-greedy exploration, where the agent takes random actions with probability ε and the best-known action with probability (1-ε).

Learning Rate (α): This parameter determines how much the agent updates its Q-values after each action. A smaller α makes learning more stable but slower, while a larger α can lead to faster convergence but may be less stable.

Discount Factor (γ): γ determines the importance of future rewards. It balances the agent's focus between immediate and long-term rewards. In Bomberman, γ is chosen to emphasize the importance of surviving and collecting coins.

Q-Table: Q-Learning uses a Q-table to store Q-values for each state-action pair. The Q-table is initially filled with arbitrary values, and the Q-learning algorithm iteratively updates these Q-values based on the agent's experiences and interactions with the environment. The Q-value of a state-action pair is updated using the Bellman equation, which incorporates the immediate reward received and the estimated future rewards from the resulting state.

3.2 Proximal Policy Optimization (Jiufeng. Li)

1. Algorithm Overview:

Proximal Policy Optimization (PPO) is a reinforcement learning algorithm designed to optimize the policy of an agent in a way that is both sample-efficient and stable. It was introduced by OpenAI and has become one of the popular algorithms for training deep reinforcement learning agents.

2. Design Process:

Policy-Based Method: PPO is a policy optimization method, which means it focuses on directly learning the policy that maps states to actions. Unlike value-based methods (like Q-learning) that estimate the value of state-action pairs, PPO aims to find a policy that maximizes the expected cumulative reward. Therefore, when designing a PPO agent, we focus on learning state-to-action strategies that maximize the cumulative reward possible.

Trust Region Optimization: PPO operates within a trust region framework, which means it constrains the updates to the policy to ensure that the new policy remains close to the old policy. This constraint helps stabilize training and prevents large policy updates that can lead to catastrophic performance drops.

Objective Function: PPO optimizes an objective function that combines the advantages of both the old and new policies. The objective is to maximize the expected improvement in policy while staying within the trust region. The key terms in the objective function are Surrogate Objective and Clipping.

Multiple Epochs: PPO often employs multiple iterations (or epochs) of collecting data from the environment, optimizing the surrogate objective, and updating the policy. This helps in stabilizing training and improving sample efficiency.

Value Function: In addition to optimizing the policy, PPO can also use a value function (like a baseline) to estimate the expected return from a state. The advantage, which is the difference between the estimated return and the expected return, is used in the surrogate objective to guide policy updates.

Exploration: PPO encourages exploration through the use of the clipping mechanism and the surrogate objective. It balances exploration and exploitation to find a good policy.

3.3 Deep Q-Network (DQN) (Jiufeng. Li)

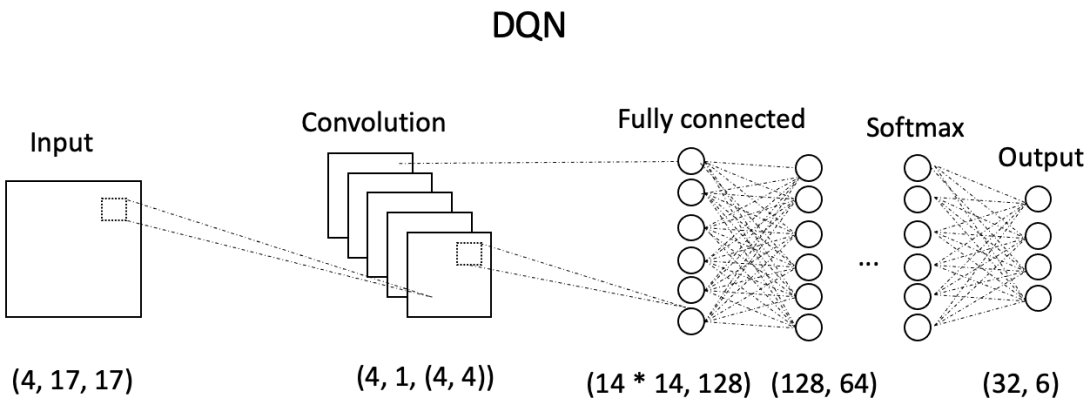
1. Algorithm Overview:

DQN is a deep reinforcement learning method that extends Q-Learning to high-dimensional state spaces, making it suitable for problems like Bomberman. Instead of using a Q-table, DQN employs a neural network (Q-network) to approximate Q-values.

2. Design Process:

State Representation: DQN can handle high-dimensional state spaces, such as game board images. We use a convolutional neural network (CNN) to process game borad images. Then the features are extracted, in our DQN model, there are four main feature channels. And those four feature matrices are used as input channels (game board, own and opponents' locations , wall location, and location of bombs and dangerous areas).

Network:



There is a 2D convolutional layer with 4 input channels, 1 output channel and a 4x4 kernel size. It is used to process the input state, which presumably has 4 channels (game feature channels). Then there are three fully connected (linear) layers. ReLU activation follows after each fully connected layer. Rectified Linear Unit (ReLU) activation function, applied after the first linear layer and subsequent linear layers. Softmax activation is applied to the output layer, which is used to convert raw values into a probability distribution over the action space

Experience Replay: Experience replay is a technique commonly used in deep reinforcement learning, particularly in Q-learning and its variants like Deep Q-Networks (DQN). It involves storing and reusing a collection of past experiences (state, action, reward, next state) in a replay buffer while training an agent. This buffer allows experiences to be randomly sampled during the training process. During the training process, The sampled batch of experiences is used to update the agent's Q-network (neural network). The Q-network learns to approximate the Q-values, which represent the expected cumulative rewards for taking different actions in different states.

Loss Function & Optimization: The loss function used during training, typically the mean squared error (MSE), measures the difference between predicted Q-values and target Q-values. We selected Adam optimizer to update the Q-network's weights based on the loss.

Hyperparameters: There are some important hyperparameters such as the learning rate, batch size, replay buffer size, and the frequency of target network updates. These hyperparameters impact training stability and convergence speed.

Action Selection: During inference, the action selection strategy depends on the trained Q-network. Typically, the action with the highest predicted Q-value is chosen.

Training Process: Training DQN involves repeatedly interacting with the environment, storing experiences, and updating the Q-network's weights to minimize the loss. The whole training process of course also involves the adjustment of the reward, the reward function has a very important role in the training of the whole network, we have different reward for different actions in the process of training the model.

4. Feature Engineering (C. Zehender)

As already described in the previous chapter, the huge amount of possible states makes it necessary to condense the most important information about the field into an overseeable number of features. Otherwise, this high dimensionality of the state space can make learning very difficult, as the agent must explore a vast number of different states. Feature engineering helps reduce dimensionality by selecting or creating relevant features.

Since the focus of this report is on machine learning approaches, explaining the details and calculations behind the features would go beyond the scope of the report. Therefore, we will focus on the importance and aim of our features rather than how we created them:

Not all aspects of the environment are equally relevant for decision-making. We aimed to identify and keep the most critical aspects of the states that are directly related to good performance, and we created more abstract features to simplify the learning problem. This abstraction can, in theory, be increased until one has a nearly rule based agent.

The overall task of winning the game can be divided into the tasks of surviving, collecting coins, and hunting other agents in the rather complex “classic” environment with crates and bombs. This means our agent needs to know where the most relevant (e.g. the closest one) of those objects are. We therefore gave our agent the x and y coordinates of the closest coin, closest crate, closest agent and closest bomb relative to its own position on the field. Since there are more than one coin, bomb and other agent, we considered it relevant to give our agent information about the rest coins, rest bombs and rest agents as well to allow it to figure out ideal strategies on its own (e.g. collecting coins as fast as possible). With this in mind, we calculated the x and y coordinates of the unweighted centres of gravity of the rest of the coins, bombs and agents relative to our agents’ position as well as their mean distances to said centres to give our agent some kind of peripheral sense. That way, our agent should have enough information to keep track of all relevant objects for survival and scoring.

One problem with those object related features is that those objects need to exist to return meaningful values (e.g. there are no coordinates for the closest coin without coins). We handled this problem by marking those cases with NaN values and dealing with them within the neural network (See below under Network design).

Besides object related features, the field itself is a central part of the game since crates limit the possibility of reaching coins and other agents and at the same time limit one's ability to flee from bombs. This emphasizes the need for some kind of local awareness. After some initial tries where we gave our agent different values for the 4 surrounding tiles depending on what was on that tile (e.g. 1 for coin, 0 for free, -4 for explosion, etc.), we created a feature that returned 0 if the tile is safe and 1 if the tile is dangerous or occupied (stone, crate, other agent, explosion). And we included the current tile in the features. Dangerous in the context of this function means that we check for the timer of currently ticking bombs, and if the agent has to move now to still have the possibility of getting out of the explosion area without hiding behind a stone, we consider a tile dangerous. Stepping onto a dangerous field is therefore not a death sentence, but it increases the likelihood of dying.

Since the field is point symmetrical to its centre and especially corners limit the possibility of fleeing, we gave our agent the distance to the wall bzw. centre of the field as a feature to give it an idea of its position. After observing that our agent tried several times to drop bombs without having the possibility to do so, we also included the bomb_possibility in our features.

This summed up to 25 features in total.

After starting to create a multi-branch network, we created a list of input tensors, with each tensor of that list supposed to go into its own branch. Those tensors need to be the same size. Therefore, we created another feature. It calculates, without taking walls, stones or crates into account, the number of possible ways to die by giving the number of tiles that are in the explosion area of currently active bombs and can be reached within the timer of said bombs. This number was calculated for our own agent, the nearest agent and in average for the rest of the agents. Making 28 features or 7 features for 4 branches (more below).

5. Training framework

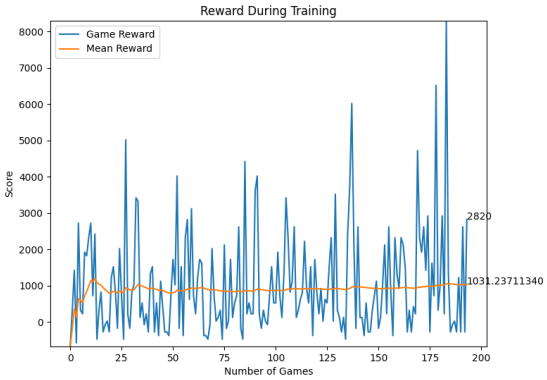
5.1 Performance tracking (C. Zehender)

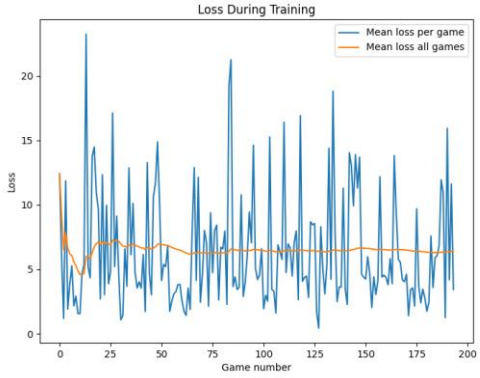
Performance tracking is an important part of training a model by systematically monitoring metrics such as loss, convergence rates and real world performance (represented by rewards). One can therefore gain valuable insights into the model's behaviour and its ability to generalize to unseen data, which is the foundation for all subsequent optimization of the model (e.g. hyperparameter tuning, network design, feature design, etc.). It also helps to identify potential issues, such as overfitting or underfitting, and is therefore a central part of the debugging process. As a last aspect, performance tracking is also part of the documentation of the models' development process.

In our case, the average loss and the total reward per game, as well as the total number of steps, record, mean reward and mean loss in the current training scenario, were saved into a txt.-file. The same data was used to plot the loss and reward per game and the average loss and mean reward over the number of played/trained on games. Since for exploration purposes we introduced a random element into the training, the performance in an individual game does not contain a lot of insight, but the mean over several games mirrors the model's current performance quite well.

Exemplary output for performance tracking:

```
Command main.py play --no-gui --agents abstract_agent rule_based_agent rule_based_agent --train 1 --scenario classic --n-rounds 20001
Game 1, Game Loss 12.409409523010254, Mean Loss 12.409409523010254, Actions 11, Mean reward -680.0, Reward -680, Record -680
Game 2, Game Loss 5.924133777618408, Mean Loss 9.16677188873291, Actions 21, Mean reward -180.0, Reward 320, Record 320
Game 3, Game Loss 1.183545708656311, Mean Loss 6.505696773529053, Actions 28, Mean reward 353.3333333333333, Reward 1420, Record 1420
Game 4, Game Loss 11.863369941711426, Mean Loss 7.845114707946777, Actions 8, Mean reward 120.0, Reward -580, Record 1420
Game 5, Game Loss 1.908318042755127, Mean Loss 6.657755374908447, Actions 49, Mean reward 640.0, Reward 2720, Record 2720
Game 6, Game Loss 3.8338706493377686, Mean Loss 6.187107563018799, Actions 31, Mean reward 586.6666666666666, Reward 320, Record 2720
Game 7, Game Loss 5.272091865539551, Mean Loss 6.05639123916626, Actions 20, Mean reward 534.2857142857143, Reward 220, Record 2720
```



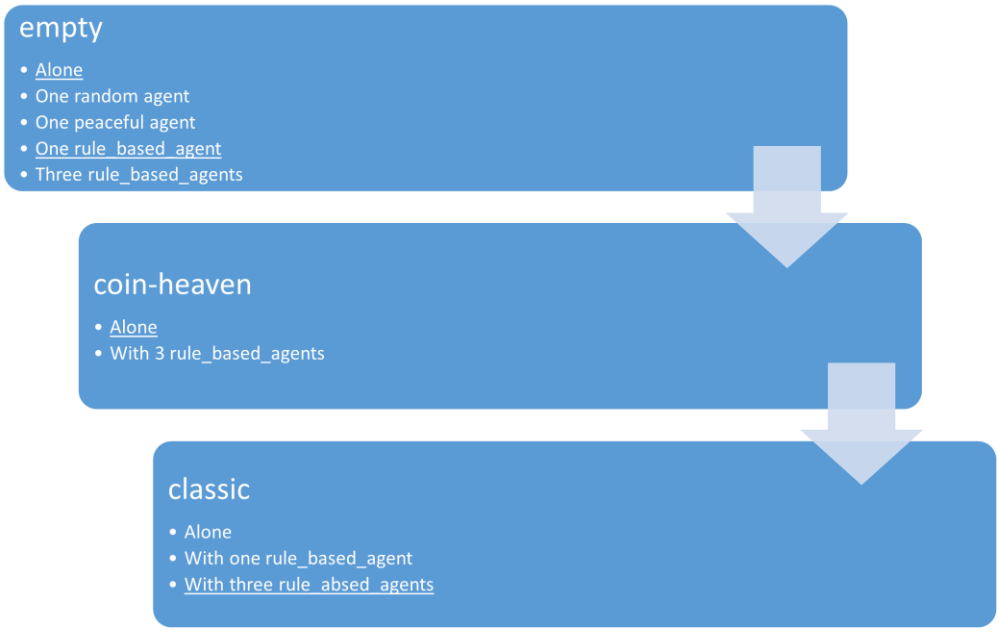


5.2 Curriculum learning (C. Zehender)

Curriculum learning draws inspiration from human learning, where learners typically start with simpler concepts and gradually progress to more complex ones. The concept was popularized by Bengio et al. in their 2009 paper "Curriculum Learning," where they showed that training deep neural networks with progressively more challenging examples can lead to significant improvements in performance.

The idea is that by gradually increasing the difficulty of training examples, curriculum learning encourages models to learn a hierarchy of features and helps them avoid getting stuck in local optima. This should result in the model converging faster and achieving better final performance (Bengio 2009):

Originally planned curriculum learning strategy:



The original idea was to train our final model according to the above seen detailed curriculum learning strategy, starting with a simple environment and then gradually increasing the complexity of the task. Due to time issues, in the end we had to go with a simpler plan consisting of the underlined parts in the above diagram.

Since not dying was considered by us to be the most central part of good performance, we planned on training the model to survive the game first. Therefore, it first had to learn not to kill itself. For this purpose, we disabled in "settings" the code that ends the game after all coins are collected and no other agents are alive and trainedour agent according to our self-defined rewards bzw. with imitation learning. Thereafter, we introduced additional agents.

After survival is secured, the agent should learn how to collect coins, ideally while standing in concurrence with other agents.

As the most complex tasks, survival and coin-collection in the classic-environment were considered. We at first didn't want to implement pathfinding algorithms (e.g. Dijkstra's algorithm) since we considered them too close to a rule_based agent. This self-expectation was, at the very end, a little overthrown after hearing that other groups used way more abstract features than ours and facing the unsatisfying performance of our models. Nonetheless, at this point, our agent had to figure out on its own how to deal with blocked paths and the in general, more complex environment with the features described above. So as a last step, the agent was trained in the final "classic"-environment.

5.3 Exploration vs. Exploitation (Jiufeng. Li)

Exploitation is defined as a greedy approach in which agents try to get more rewards by using estimated value but not the actual value. So, in this technique, agents make the best decision based on current information. Unlike exploitation, in exploration techniques, agents primarily focus on improving their knowledge about each action instead of getting more rewards so that they can get long-term benefits. So, in exploration, agents work on gathering more information to make the best overall decision.

However, both of these techniques are not feasible simultaneously, but this issue can be resolved by using Epsilon Greedy policy. Epsilon greedy policy is defined as a technique to maintain a balance between exploitation and exploration. However, to choose between exploration and exploitation, a very simple method is to select randomly. This can be done by choosing exploitation most of the time with a little exploration.

In the greedy epsilon strategy, an exploration rate or epsilon (denoted as ε) is initially set to 1. This exploration rate defines the probability of exploring the environment by the agent rather than exploiting it. It also ensures that the agent will start by exploring the environment with ε=1.

As the agent start and learns more about the environment, the epsilon decreases by some rate in the defined rate, so the likelihood of exploration becomes less and less probable as the agent learns more and more about the environment. In such a case, the agent becomes greedy for exploiting the environment.

Epsilon-Greedy Action Selection

$$\text{Action at Time(t)} = \begin{cases} \max Q_t(a) & \text{with probability } 1-\epsilon \\ \text{any action(a)} & \text{with probability } \epsilon \end{cases}$$

To find if the agent will select exploration or exploitation at each step, we generate a random number between 0 and 1 and compare it to the epsilon. If this random number is greater than ε, then the next action would be decided by the exploitation method. Else it must be exploration. In the case of exploitation, the agent will take action with the highest Q-value for the current state.

5.4 Loss calculation (C. Zehender)

For Deep Q-learning as well as Proximal Policy Optimization the Mean Squared Error (MSE) is a common choice for the loss calculation and was our first choice too. In both DQL and PPO, the goal is to estimate a value function (Q-values in Q-learning and state-value or advantage functions in PPO) that represents the expected cumulative rewards. The MSE then measures the squared difference between the predicted and target values.

The MSE therefore provides a clear and intuitive interpretation: It penalizes larger errors more heavily than smaller errors, since larger deviations from the target result in larger loss values.

Besides that, the MSE is a continuous and differentiable loss function, which allows for efficient computation of gradients during the backward pass.

Only one problem exists with the MSE: It is quite sensitive to outliers in the data due to the squaring operation. Since we introduced a random element for exploration, this is a not desirable flaw.

We therefore decided to go with the Smooth Absolute Error Loss/Huber Loss (Smooth L1 Loss): The Smooth L1 Loss is a combination of the Absolute Error (L1 loss) and MSE loss. It uses a piecewise function that behaves linearly for small errors, similar to those of the L1 loss, and quadratically for larger errors, similar to the MSE loss. This gives it the desirable characteristic of being less susceptible to outliers. The Smooth L1 Loss is defined as following:

$$L(x) = \begin{cases} 0.5 \cdot x^2, & if\ x < |\alpha| \\ |x| - 0.5 \cdot \alpha^2, & otherwise \end{cases}$$

α is a hyperparameter that determines the point at which the loss transitions from quadratic to linear behaviour. (Fitzgibbon 2001)

5.5 Single step vs. batch updates (C. Zehender)

Often the standard approach in reinforcement learning is the use of batch updates, since it has several advantages over the use of single-step updates:

While single step updates rely on individual experiences, batch updates aggregate experiences from multiple steps. This means that batch updates typically have lower variance in the estimate of the

value function, which usually leads to more stable learning and faster convergence. Especially when introducing a random element batch updates can drastically outperform single step updates.

On the other hand batch updates require a memory buffer (in this case useful anyway, see below) and may be in simple and stable environments with naturally low variance (e.g. with very abstract, highly processed features) outperformed by single step updates.

We tried out single step updates and batch updates for most of our agents and observed the above mentioned trend: The more complex/less abstract the features are the bigger is the advantage of batch updates over single step updates.

5.6 Memory replay (C. Zehender)

The concept of memory replay in reinforcement learning has its origins in the field of experience replay, which was introduced as a key component of the DQN algorithm by Volodymyr Mnih et al. in their 2015 paper "Human-level control through deep reinforcement learning."

The core idea of experience replay is to store experiences (e.g. old_state, action, reward, new_state) in a replay buffer and sample mini-batches of experiences during training. These mini-batches are used to update the neural network representing the value function (e.g. Q-values in Q-learning).

On the one hand, memory replay allows for the reuse of past experiences, reducing the need for fresh interactions with the environment. And on the other hand, in reinforcement learning, consecutive experiences are often correlated in time, which can lead to unstable learning (e.g. bombs explode after 4 steps). Memory replay breaks these temporal correlations by randomly sampling experiences from the buffer, improving the quality of the training data.

It also helps the agent escape after getting stuck in suboptimal policies by learning from previous (potentially better) experiences. But this can also hinder training if the replay buffer is too big and repeatedly presents very old experiences. The size of the replay buffer is therefore another hyperparameter.

To conclude, by training on diverse past experiences rather than just the most recent transitions using memory replay, our agent achieves more stable learning and improved performance. Besides, another advantage of establishing a replay buffer is that the reward for past events can be modified (e.g. bombs have an effect after 4 steps) and batch updates can be established. (Mnih 2015)

5.7 Reward shaping

5.7.1 Reward normalization (C. Zehender)

Reward normalization is a technique commonly used in reinforcement learning to address challenges related to reward scaling and variance. It aims to create a consistent and stable learning environment for agents when dealing with environments that provide rewards on varying scales.

Reward normalization helps make the learning process more stable. By scaling rewards to have a consistent range, it prevents excessively large or small rewards from dominating the learning process and getting stuck in said states. With normalized rewards, the agent also generalizes better to different tasks and environments (e.g in different steps of the curriculum learning). It ensures that the learned policy is not overly specific to the scale of rewards in a particular environment, making it more transferable.

Although there are many ways to normalize rewards, we went with Z-score normalization. With a scaling_factor of 1, it scales rewards to have a mean of 0 and a standard deviation of 1. The formula for our z-score normalization is:

$$Normalized\_Reward = \frac{Reward - Mean\_Reward}{(Standard\_deviation\_of\_the\_rewards + \varepsilon) \cdot scaling\_factor}$$

In above formula, ε is a small positive value (e.g. 0.0000001) to avoid division by zero, and scaling_factor is another hyperparameter. It prevents over-normalization by allowing a wider spread of normalized rewards compared to a standard z-score normalization. If one normalizes the rewards too aggressively, it can lead to problems if the original rewards have a wide range. Then one might end up squeezing the rewards into a very narrow range, which can make all actions appear equally good or bad for many ML algorithms, including DQL, causing the loss to converge to zero prematurely.

5.7.2 Reward-based imitation learning vs. Self-customized reward functions (C. Zehender)

For our reward system we explored two different approaches:

First we implemented a reward-based imitation learning strategy by simply copying the code of the rule_based_agent in our training function and handing out a positive reward of +100 if our agent took the same action or a negative reward of -100 if our agent took a different action from the rule_based_agent.

We then compared this first approach with the performance of our agent if we implemented self-customized reward functions. The already

Reward table:

event/action	current action	current_action -1	current_action -2	current_action -3	current_action -4	current_action -5	current_action -6	current_action -7	current_action -8	current_action -9	Additional requirements
killed_by_explosion	-40	-	-	-	-	-	-	-	-	-	
killed_self	+20 (if moved away from bomb) -40 (if not moved away from bomb)	+20 (if moved away from bomb) -40 (if not moved away from bomb)	+20 (if moved away from bomb) -40 (if not moved away from bomb)	+20 (if moved away from bomb) -40 (if not moved away from bomb)	+20 (if moved away from bomb) -40 (if not moved away from bomb)	-	-	-	-	-	The agent must not move onto a dangerous field or stay on one.
killed_by_opponent	+32 (if moved away from bomb) -52 (if not moved away from bomb)	+32 (if moved away from bomb) -52 (if not moved away from bomb)	+32 (if moved away from bomb) -52 (if not moved away from bomb)	+32 (if moved away from bomb) -52 (if not moved away from bomb)	+32 (if moved away from bomb) -52 (if not moved away from bomb)	-	-	-	-	-	The agent must not move onto a dangerous field or stay on one.
coin_collected	see description below										
bomb_dropped	+32 (if moved towards the closest or mean rest agent)	+32 (if moved towards the closest or mean rest agent)	+32 (if moved towards the closest or mean rest agent)	+32 (if moved towards the closest or mean rest agent)	+32 (if moved towards the closest or mean rest agent)	+32 (if moved towards the closest or mean rest agent)	-	-	-	-	At least one agent must be within 3 tiles. The more agents are within 3 tiles the higher the reward
bomb_exploded	+42	+42	+42	+42	+42	-	-	-	-	-	Agent must be at least within 4 tiles of the bomb before it explodes and moves away from it or moves onto a save field
killed_opponent	+30	+30	+30	+30	+30	+48	+24	+24	+24	-	The agent must not be on a dangerous field when dropping the bomb
crate_destroyed	-	-	-	-	-	+24	+12 (moved toward the closest crate)	+12 (moved toward the closest crate)	+12 (moved toward the closest crate)	+12 (moved toward the closest crate)	
coin_detected	-	-	-	-	-	+24	+12 (moved toward the closest crate)	+12 (moved toward the closest crate)	+12 (moved toward the closest crate)	+12 (moved toward the closest crate)	

A table that describes how rewards are handed out depending on the occurred event.

provided possible rewards proved to be insufficient quite fast. Not only didn't allow this function to request additional requirements (e.g. don't move towards a ticking bomb to collect a coin), it also didn't pay attention to the time dependency of some rewards (e.g. crate_destroyed is an effect of dropping a bomb 5 steps previously). We therefore used our reply buffer to update the reward of a previous action if it turned out to have affected the game positively or negatively. Going through all

customized reward functions in detail would go beyond the scope of this report, but in below table are the handed out rewards depending on the occurred event.

One special reward function, the train_coin_collected-function should be described more in detail:

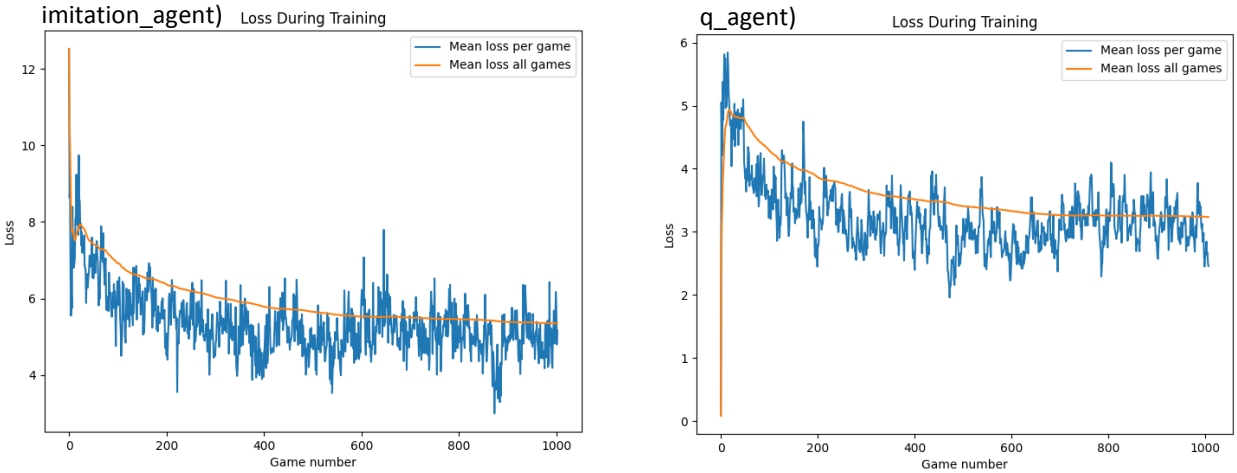
Regarding the collection of the coins we identified a variant of the Traveling Salesman Problem (TSP), or more specific the open TSP with Time Windows. Due to standing in concurrence with other agents a coin will be collected at some point from another agent (the closer another agent, the smaller the time window till it is collected by the other agent), but our agent has not the obligation to return to its starting point. While it is a challenging problem, there are heuristic and exact algorithms that can be applied to find approximate solutions, but since we didn't want to implement a rule based agent we didn't look further into it.

We rather designed the reward function for coin collection in a way that the reward for collecting a coin is bigger the faster a coin is collected in total and in regard to the previously collected coin. We implemented the following formula for this purpose:

$$\text{reward} = \frac{\text{total\_number\_of\_coins} \; - (10 - e^{-0.1 \cdot (\text{current\_number\_of\_collected\_coins} - 22.97)})}{2.5 - e^{-0.075 \cdot (\text{steps\_to\_the\_prevous\_collected\_coin} - 6.4)}} \cdot \frac{\text{max\_reward}}{\text{total\_number\_of\_coins}}$$

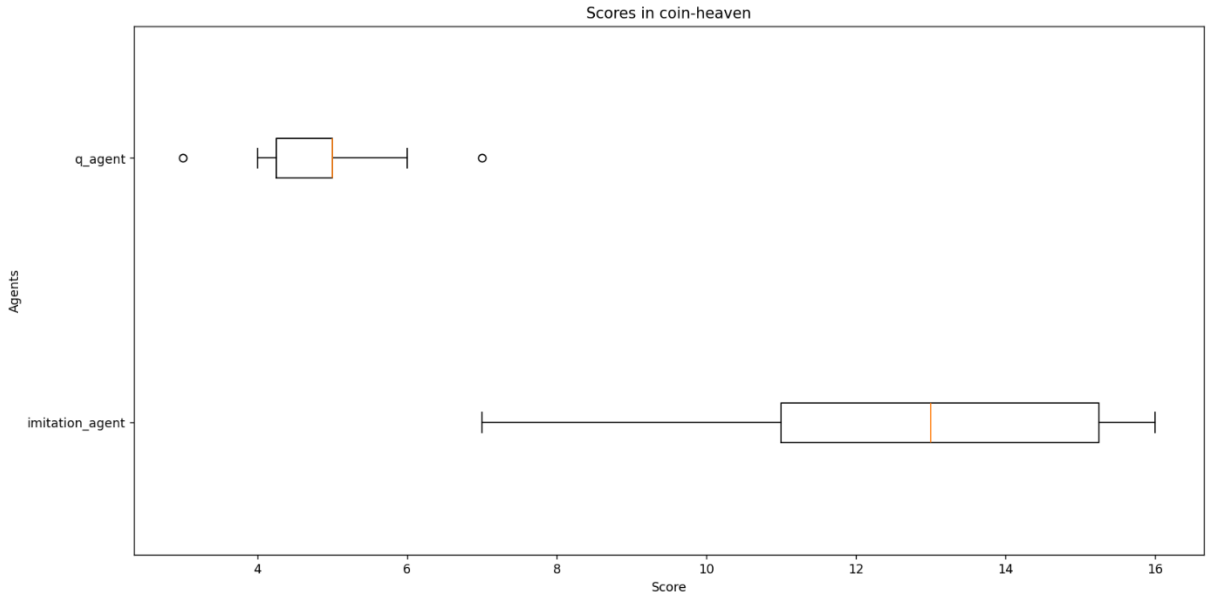
With $\text{max\_reward} = 75$. The scaling in the formula ensures a reasonable decrease in the reward while collecting a coin still gives at least a reward of 24 for all actions that brought the agent closer to the collected coin since the last one was collected without jeopardizing itself. By applying this formula the agent gets the highest reward by first collecting coins close to each other before collecting the rest coins. This should give it the incentive to collect coins in the fastest possible way.

After establishing both reward principles we trained two similarly designed models under their respective strategy and compared their performance:



This diagram compares the the loss per game during training of the imitation_agent (imitation learning) and q_agent (self-defined rewards) trained on batch updates and a multi-branch net with a random_vec of 0.25 under the command main.py play --no-gui --agents simple_net_q rule_based_agent --train 1 --scenario coin-heaven --n-rounds 20001.

Since both agents are trained with different reward functions, the absolute reward cannot be directly compared, but it is obvious for both agents that the loss is decreasing over time suggesting an adaption to the required task. To compare real world performance in this case we have to look at the absolute score achieved by those agents. In our case for 10 games in the coin-heaven scenario:



As seen in the above boxplots, both agent's performance after more than 1000 games is quite different according to their score. Besides, when observing the patterns and reactions of the agents visually are different as well. In the end, the agent based on imitation learning is a lot more advanced. It more often goes in the middle of the field and drops bombs when the other agent is near. It also more often successfully avoids the other agent's bombs. We therefore decided that, although we had put a lot of work into creating advanced and logic reward functions, to continue with reward-based imitation learning.

6. Network design

6.1 NaN Handling and Handling of zeros (C. Zehender)

As mentioned above in the feature design chapter, we needed to handle situations where our features wouldn't make sense (e.g. the coordinates of the closest coin in a scenario without coins). For those situations, we decided to mark those values as NaN-values and handle them within the net. Since we decided to apply the Smooth L1 Loss to our model, we had to replace the NaNs with numerical data (e.g. zeros). But replacing NaNs with artificial values can impact how the model learns and generalizes from the data. Essentially, the network will be penalized for the discrepancies between the masked NaNs and the true target values. Replacing NaNs with zeros and small values can introduce artificial patterns in the data that don't exist in the real world. This can lead to model overfitting, where the model learns these patterns as if they were genuine, resulting in poor generalization. Therefore, we tested three approaches for their performance:

1. Masking NaNs as zeros and keeping the valid zeros:
   One classic approach is to mask NaNs as zeros within the network. To keep in mind, since we originally used linear layers with ReLU activation, if any of the inputs to the linear layers contain zeros, the corresponding outputs of the linear layers will also be zeros due to the ReLU activation. During backpropagation, gradients will not flow through these zero values, which can result in dead neurons (neurons that always output zero) and hinder the learning process. Another point is that the model contains meaningful zeros (e.g. the closest bomb after dropping one). The idea is that the other values (e.g. bomb density) indicate whether it is a masked NaN or a regular zero, and the model will learn to distinguish them.
2. Masking NaNs as zeros and zeros as small values (e.g. 0.01):

Masking zeros as small positive values can help maintain a more consistent gradient flow during training and help distinguish between true zeros and NaN-zeros. Potentially leading to smoother convergence and faster training.

3. Masking NaNs with impossible values and keeping the zeros:
   Another approach is to mask the NaNs with values that cannot occur do to the size of the field (e.g. 20s). But replacing NaNs with too large values can cause large gradients during backpropagation, which can lead to numerical instability, making training difficult or even impossible.

Since none of the above approaches showed any difference in performance, we had to decide according to our theoretical background and gut feeling. For that reason, we went with masking NaNs as zero values and zeros as small values (e.g. 0.01), to make differentiation between true zeros and NaN-zeros easier. To avoid dead neurons we used Leaky ReLU, which introduces a small, non-zero gradient for negative inputs, which prevents neurons from becoming completely inactive during training. To avoid exploding gradients during training we used gradient clipping, which scales the gradients if their norm exceeds a certain threshold. This threshold needs to be optimized. Setting it too low may hinder learning, while setting it too high may not effectively prevent exploding gradients. From both aspects, especially using Leaky ReLU turned out to be a key factor for adequate learning.

If we had had more time, we would have explored alternative strategies, such as developing custom loss functions that handle NaNs differently to ensure accurate model training and better generalization.

6.2 Single-branch network vs. Multi-branch network (C. Zehender)

In terms of the design of the neural net, we started out with a single-branch neural net consisting of an adjustable number of linear, fully connected layers (Fully connected=each neuron is connected to each neuron of the previous layer via a linear transformation). Single-branch networks are simpler to design and train, making them a good choice for straightforward tasks, but they may struggle to effectively merge and process diverse input features. This potentially limits their ability to capture complex patterns and makes them a potential weakness for training in the evtl. rather complex Bomberman environment. To address this potential issue, we designed a multi-branch network. Those nets specialize in processing specific types of features per branch, enhancing their ability to capture complex patterns.

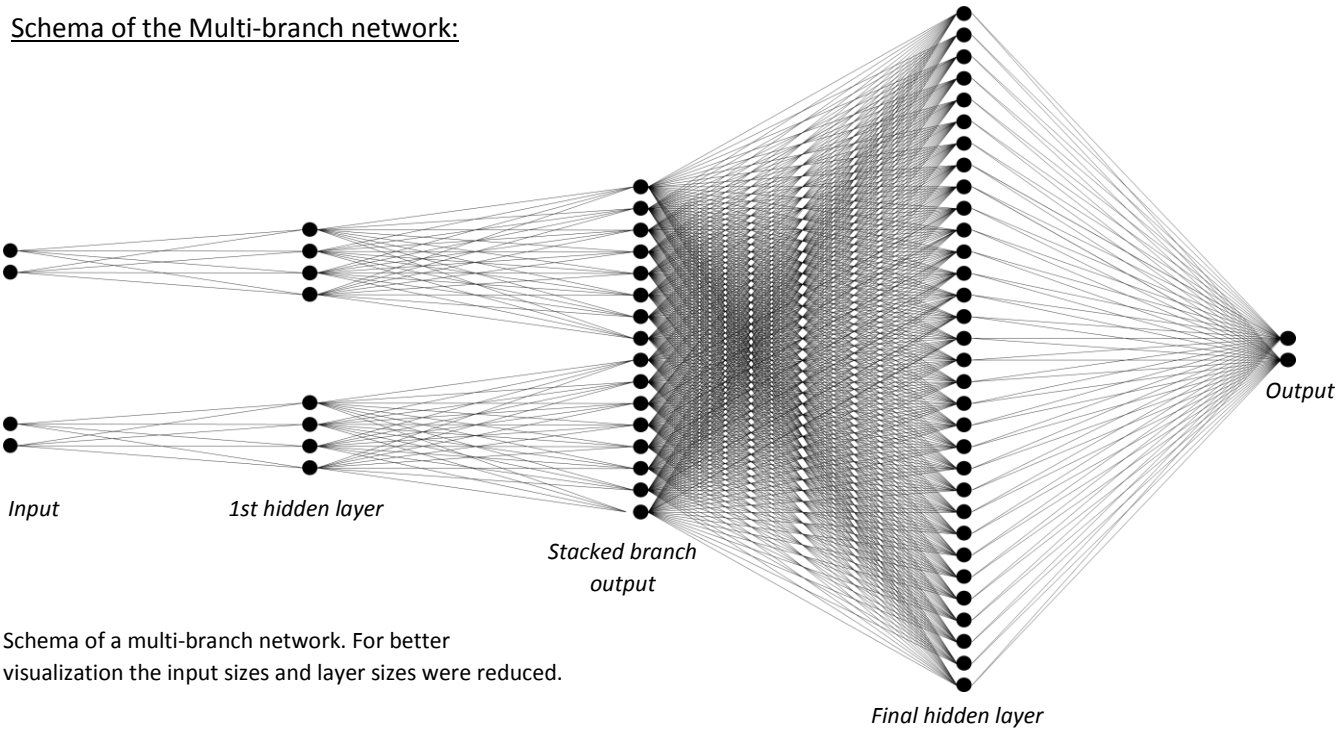
We started by designing the different branches to specialize in preprocessing specific types of features before merging said input into a final output. Therefore, we first had to categorize our 24 features into task-specific feature groups (e.g. the nearest coin for Coin collection).We identified 4 tasks and respectively 4 branches:

- Action Branch: Gets the reasonableness to move to one of the surrounding tiles or stay on the current one as binary input, the distance to the wall on the field and the possibility to drop a bomb.
- Coin collection branch: Gets the relative coordinates of the nearest crate and coin, as well as the centre of gravity and density of the rest of the coins as input.
- Survival branch: Gets the relative coordinates of the nearest bomb, its timer and the the centre of gravity and mean timer of the rest of the bombs as input. As the last and potentially the weakest feature, the rough absolute number of ways to die in the next 4 steps was given.

- Hunting branch: Gets the relative coordinates of the nearest agent as well as the centre of gravity of the rest of the agents as input. Additionally, analogous to the Survival branch, the rough absolute number of ways to die was given for the nearest agent and in average for the rest of the agents.

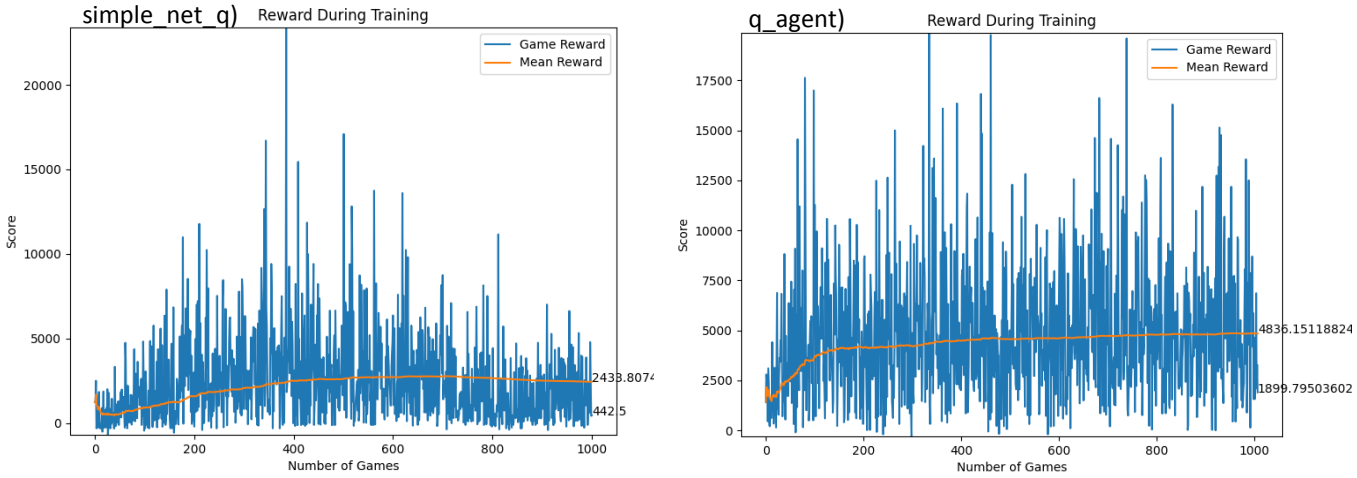
All branches preprocessed their inputs with 1 hidden linear layer of adjustable size, before the output of all branches was stacked on top of one another and processed through a last linear, fully connected hidden layer. (See the scheme of the multi-branch net below.)

Schema of the Multi-branch network:



Schema of a multi-branch network. For better visualization the input sizes and layer sizes were reduced.

Thereafter, we tested both nets for performance within similarly designed agents:



This diagram compares the the rewards per game during training of the simple_net_q (Single-branch net) and q_agent (Multi-branch net) trained on batch updates and self-defined rewards with a random_vec of 0.25 under the command main.py play --no-gui --agents simple_net_q rule_based_agent --train 1 --scenario coin-heaven --n-rounds 20001.

As seen in the above diagrams, the single-branch net is significantly outperformed by the multi-branch net in regard to reward and, respectively, the performance of the model. This may be due to the environment being complex enough that the model does benefit quite much from the complexity

of the multi-branch network. So we concluded that the task might ideally be solved by pre-processing all input features separately and continued with the multi-branch net due to its better performance.

7. Experiments and Results

As explained within the previous chapters we had decided to implement a multi-branch network along with reward based imitation learning. The only two things missing now are the implementation of the training function and the optimization of all the hyperparameters mentioned throughout the previous chapters.

7.1 Deep Q-Learning (C. Zehender)

Q-learning is a popular reinforcement learning (RL) algorithm that was developed by Chris Watkins in 1989. It is a model-free, off-policy RL algorithm that aims to find an optimal policy for an agent to maximize its cumulative reward in a Markov decision process (MDP) or a similar environment. As a model-free algorithm it doesn't require prior knowledge of the environment's dynamics or transition probabilities and learns directly from interactions with the environment. But Q-learning, in its basic form, relies on a tabular representation of Q-values, which becomes impractical for problems with continuous state spaces. This lead to the introduction of Deep Q-Learning (DQL) by Minh et al. in 2013.

The key theoretical foundation of Deep Q-learning is the use of a deep neural network to approximate the Q-function. The Q-function represents the expected cumulative future rewards for taking each action in a given state. The neural network takes the state as input and outputs Q-values for each action. The network is trained to minimize the temporal difference error between predicted Q-values and target Q-values, calculated using the Bellman equation. The Q-values represent the expected cumulative reward an agent can achieve starting from a given state, taking a specific action, and following an optimal policy thereafter.

The Bellman equation for Q-learning in general is as follows:

$$Q(s,a) = (1-a) \cdot Q(s,a) + a \cdot \left(r + \gamma \cdot max_{a'} Q(s',a')\right)$$

Where:

$Q(s,a)$ is the Q-value for state s and action a.

a is the learning rate, controlling the weight given to new information.

r is the immediate reward received after taking action a in state s.

γ is the discount factor, representing the importance of future rewards.

$max_{a'} Q(s',a')$ is the maximum Q-value for the next state s' over all possible actions a'. (Mnih 2013)

Using the above theoretical background we calculated accordingly the Smooth L1 Loss between the Q-value for the taken action and the predicted Q-values and used this value to update our model.

7.1.1 Hyperparameter optimization (C. Zehender)

Hidden layer size and number of hidden layers per branch: After some testing, we found out that after one hidden layer per branch with 14 neurons, the quality didn’t improve reasonably compared to the increased training time.

Final hidden layer size and number of hidden layers: Similarly, after some testing, we found out that after one hidden layer before the final output with 168 neurons, the quality didn't improve reasonably compared to the increased training time.

Batch size of batch updates: Since we decided to use a training strategy with the need for a replay buffer, we had to choose a batch-size for sampling said one. The batch was randomly chosen from the experience buffer. We tried out batch sizes between 10 and 100 actions and came to the conclusion that larger batch sizes slowed down the training progress without improving the convergence speed or the quality of the result reasonably. In the end, we decided to use a batch size of 16 for most games.

In hindsight, implementing a function that would have prioritized actions with especially positive rewards or other criteria probably would have sped up learning and improved the performance of the model.

Maximal size of replay buffer: At first, we had a very large replay buffer with 2000 to 4000 previous actions where we took random samples. But after realizing that our agent was alive for an average of just 40 actions in most settings, we decided to stay with a memory size of 400 actions, which equals roughly the last 10 games (more or less, depending on the training setting).

Update steps of our target_model: We experimented with different lengths of periodic updating our target_model. While large periods (e.g., updating every 100^{th} game) seemed to slow down convergence, small periods (e.g., updating every game) seemed to be an unnecessary computational burden. At the end, we decided to update our target network every 10^{th} game.

Maximal size of normalization buffer: For the size of the normalization buffer used for the calculation of the standard deviation and mean of previous rewards, we used a size of 250 previous rewards. The aim was to have a reward normalization adapted to the current setting and strategy; therefore, it shouldn't be too big, but at the same time big enough to fulfill its purpose of normalization. The size of 250 worked quite well for us.

random_prob of the ε –greedy strategy: As we used the simple, yet effective ε –greedy strategy to control our agents trade-off between exploration and exploitation we had to determine the starting point of our random_prob bzw. ε, as well as the probability of every action. It doesn't make sense to keep those values static over the entire training period, since the agent learns and gains more knowledge, making excessive exploration unnecessary. So in later training steps, the ε should be reduced to prioritize exploitation. We decided to start with a random_prob of 0.25 and reduce it continuously by hand in later training steps down to 0.01.

The probabilities for every action were adapted to training environment and aim (e.g. for alone in "coin-heaven" we decided for a lower probability of "wait" and "bomb").

Scaling factor for normalization: Most of the time, a scaling factor of 0.1 was applied. If the loss of the individual steps showed low variance within our txt.-output file, we decreased it to 0.01, which worked in most cases for us.

Threshold for gradient clipping: We went with a clipping factor of 1, which worked quite well for us without further optimization.

Slope of Leaky ReLU: A common choice for the slope of LeakyReLU is a small positive value. We went with 0.01, which worked well for our setting. Slightly higher or lower slopes didn't change much about the training performance.

Learning rate: During the different steps of the agent's development, we tested several values of the learning rate between $0.1 > \alpha > 0.0001$. However, we found a learning rate of around 0.01 worked best for our setting. For the other learning rates, we received worse results after the training or too slow convergence for our purposes.

γ –value bzw. discount factor: Optimizing the discount factor was a little tricky. After testing out the whole range from 0 to 1.5, we saw that our model learned best at lower discount factors of around 0.15.

Since, the discount factor determines the importance of future rewards in the agent's decision-making process, it influences how much weight the algorithm gives to immediate rewards compared to future rewards. Since our rewards are based on actions in the current state space and all future rewards are already taken into account, it makes sense that our model places more importance on short term rewards.

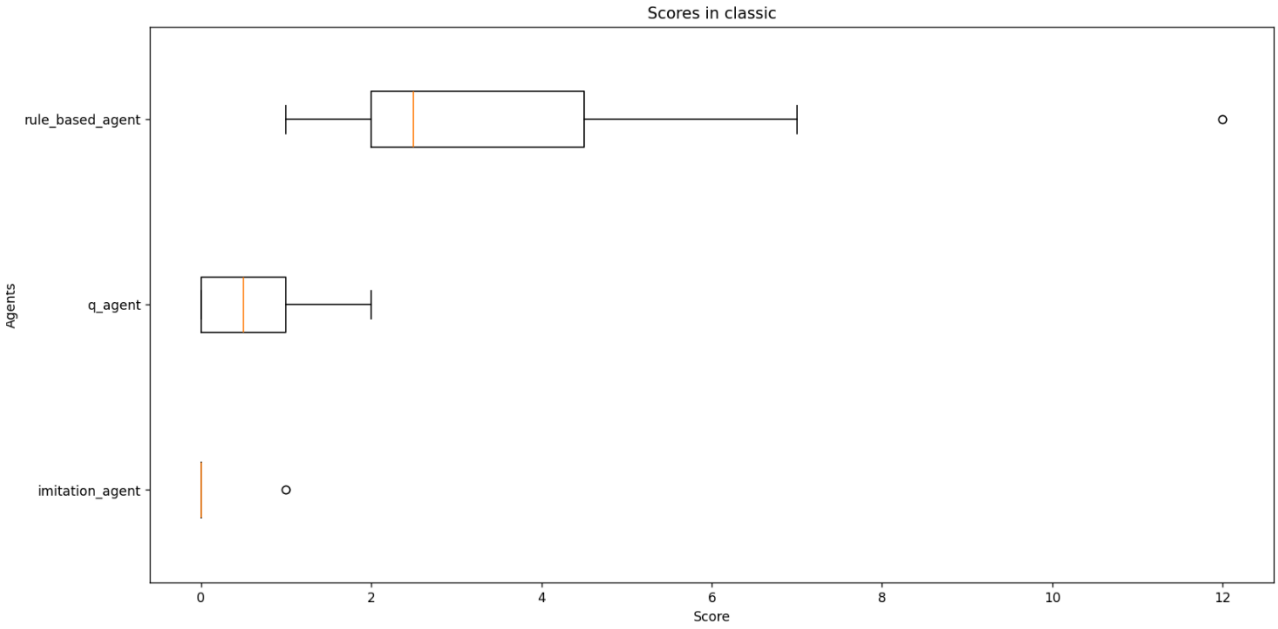
Optimizer choice: As an optimizer, we decided to use Adam in its default settings. Adam was introduced by D. P. Kingma and J. Ba in their 2014 paper titled "Adam: A Method for Stochastic Optimization" It combines ideas from two other popular optimization algorithms—RMSprop and Momentum. A more detailed look at the theoretical foundation can be found in said paper.

Since Adam combines the benefits of both momentum and RMSprop, it often converges faster than traditional stochastic gradient descent (SGD) on a wide range of tasks. Adam is also less sensitive to the choice of hyperparameters (e.g. learning rate) compared to some other optimizers. This makes it easier for us to focus on optimizing the hyperparameters for other purposes (Kingma 2014).

7.1.2 Results (C. Zehender)

Since the entire setup for training and design choices was already explained in previous chapters, we will keep this chapter short and focus on the final performance in the game:

Below are displayed the performance of the imitation_agent (DQL, imitation-based), q_agent (DQL, self-defined rewards) and rule_based_agent in the classic scenario with three other rule_based_agents:



As seen in the above boxplot, although our imitation_agent performs quite well in the coin-heaven scenario, it doesn't manage to destroy crates, leading to a quite bad performance in the classic-scenario. Maybe an additional feature would have solved our problem (e.g. a check for bordering three crates or the connectivity of the current tile to the rest tiles), but we hadn't had the time to train the network with another feature and rearrange the entire multi-branch network.

Our self-defined rewards, on the other hand, worked better in the classic scenario, since they led to the model dropping bombs not just close to agents. But in the end, the overall performance of both models in the classic scenario remained quite unsatisfying.

7.2 Proximal Policy Optimization (C. Zehender)

Since the results for DQL weren't that satisfying at the beginning we shortly implemented another training function based on Proximal Policy Optimization (PPO), we abandoned quite early thereafter due to the limited time.

Proximal Policy Optimization (PPO) was introduced by researchers from OpenAI in 2017 as an improvement over the original Policy Gradient methods. PPO is built on the idea of balancing the trade-off between making significant policy updates and ensuring that the policy changes are not too extrem to maintain stability during training.

The theoretical foundation of PPO involves a trust region optimization approach. The goal of PPO is to optimize a policy, represented by a parameterized policy function $\pi_\theta(a \mid s)$, where θ are the parameters of the policy, a is an action, and s is a state. It aims to maximize the expected return bzw. cumulative reward, under said policy. The objective function for policy improvement can be written as:

$$J(\theta) = \mathbb{E}_t\left(\min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}}A_t^{clip}, clip\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}}, 1-\epsilon, 1+\epsilon\right)A_t^{clip}\right)\right)$$

where $\pi_{\theta old}$ is the policy with old parameters, and A_t^{clip} is the clipped advantage function. Besides, the use of a surrogate objective and a clipping mechanism helps create a balance between exploration and exploitation, making PPO suitable for our task. (Schulman 2017)

7.2.1 Hyperparameter optimization (C. Zehender)

We chose the same setup as under DQL without further optimization, although if we had continued with this approach, a new hyperparameter optimization for the different training function would have been advantageous.

The additional hyperparameters for PPO were optimized, though:

clip_epsilon: It establishes the threshold for the clipping mechanism in the PPO objective function. The clipping is applied to the ratio of the new policy probability to the old policy probability to ensure that policy updates are within a certain range. Since we didn't want to extreme updates, we went with a value of 0.2.

value_coeff: It is the coefficient for the value loss in the overall PPO objective function. The PPO algorithm combines a policy loss (from the policy improvement term) with a value loss term. The value_coeff determines the weight or importance assigned to the value loss relative to the policy loss. We went with a neutral approach of 0.5, which worked well for us.

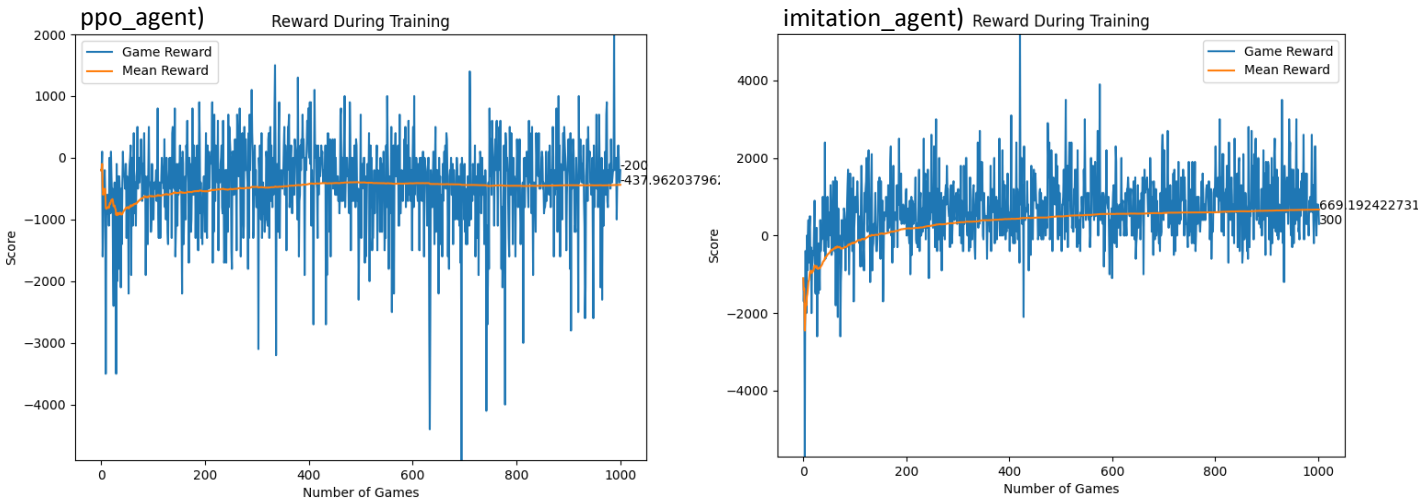
entropy_coeff: It is the coefficient for the entropy term in the overall PPO objective function. The entropy term encourages exploration by penalizing policies that are too deterministic. Higher

entropy values lead to more exploration, but since we already have an exploration strategy, we went with a lower value of 0.01.

7.2.2 Results (C. Zehender)

Again, since the setup for training and design choices was already explained in previous chapters, we will keep this chapter short and focus on the final performance in the game:

Below are displayed the performance of the ppo_agent (PPO, imitation-based) compared to the imitation_agent (DQL, imitation-based):



This diagram compares the the rewards per game during training of the ppo_agent (PPO based) and imitation_agent (DQL based) trained on batch updates and imitation based rewards with a random_vec of 0.25 under the command main.py play --no-gui --agents simple_net_q rule_based_agent rule_based_agent rule_based_agent --train 1 --scenario coin-heaven --n-rounds 20001.

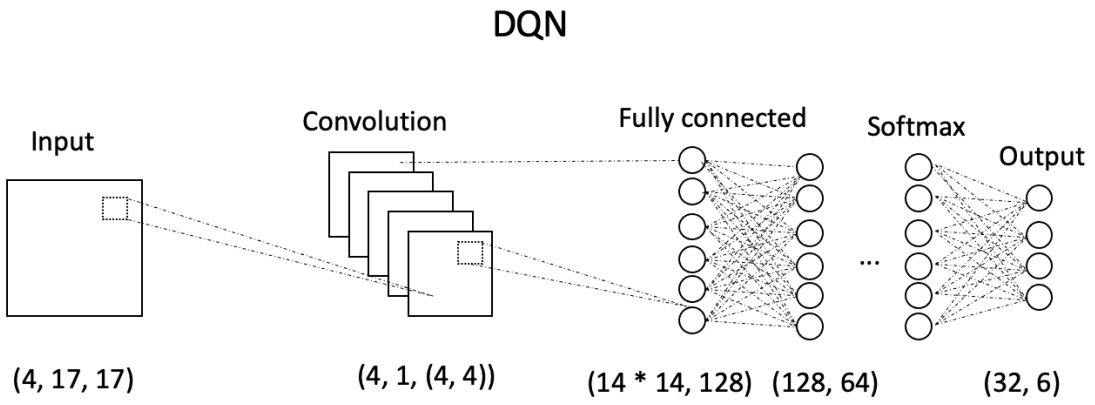
As seen in above diagrams, the DQL-based agent performs a lot better than the PPO-based agent, when comparing their achieved rewards. We got the same result when assessing the agents' performance visually and therefore decided to continue bzw. stay with the DQL-based training strategy.

7.3 Deep Q-Network with convolutional layers (Jiufeng. Li)

While Q-Learning is a foundational algorithm for reinforcement learning, it is limited to simple problems with discrete state spaces. DQN, with its deep neural network function approximation, extends the capabilities of Q-Learning to handle complex tasks with high-dimensional state spaces, making it a key advancement in reinforcement learning, especially for tasks like bomberman game using raw pixel inputs.

We tried different neural networks for training, here we read the features of the game borad by using convolutional neural network which is more intuitive and at the same time we mapped the feature dimensions to the number of channels so that each 2D game borad feature corresponds to a channel.

Here is the structure of the network:



7.3.1 Hyperparameter optimization (Jiufeng. Li)

Feature channel: We initialized four sets of feature vectors, the location of the coin as channel 0, the coordinates of the location of all the opponents as channel 1, the coordinates of the user and the opponents as channel 2, and lastly the area affected by the bomb blast as channel 3. The tuning of the input feature vectors is also very important, in our tests we found that since we are using 2D features, the input feature vectors should not be too much, which will lead to slow convergence of the training. So, we choose the four most important features.

Learning Rate (α): The learning rate controls the step size during gradient descent. It determines how much the model's weights are updated in response to the loss. Typically, a small learning rate like 0.0001 or 0.001 is a good starting point. After conducting several tests, we found that 0.005 would perform better and we would be better off choosing 0.005.

Discount Factor (γ): The discount factor controls the weight of future rewards in the Q-learning update equation. A common value is 0.99, but it depends on the task. Smaller values prioritize immediate rewards, while larger values prioritize long-term rewards. We will consider the current reward even more, because as a game with a relatively short win/loss time per game, focusing on the current state of the game will be more favorable for victory. So after many tests, we found that Discount Factor is better to take 0.99.

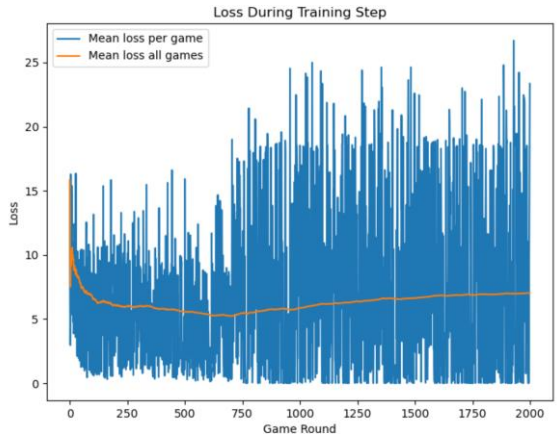
Batch Size: The batch size determines how many experiences are used in each gradient update step. A larger batch size can lead to more stable updates but may require more memory. Finally we take 16.

Replay Buffer Size: The replay buffer stores past experiences for training. A larger buffer can help stabilize training by reducing sample correlations. For with convolutional DQN, we initialize the replay buffer to 1000.

Loss Function: The choice of loss function can affect convergence. At last, we choose Mean Squared Error (MSE) loss function. It performs pretty well at model convergence.

Optimizer: We Use Adam optimizer which is a popular choice due to its adaptive learning rate.

7.3.2 Results (Jiufeng. Li)



Through the above figure, we can analyze that the recognition of input features through convolutional layers does not play a big advantage compared to simple net. The model convergence is relatively slow because the feature vector dimension becomes two-dimensional, and the average reward does not have a significant improvement with the increase of step.

7.4 Debugging and agents with highly abstract features (C. Zehender)

Besides regular debugging of logical errors and code problems, we also wanted to find a way to test if our features and training strategy is able to train a well performing model.

For that purpose we used our features to create a rule based agent, called rule_agent. We just implemented a few more, very simple decision trees and weighted the importance of our features by hand (similar to the given normalized rewards). Then we multiplied the inverse binary reasonable_action_branch without distance to the wall with our self-customized multipliers derived from the features of the other branches and that way determined the most reasonable action in the given situation by taking the action with the highest score. After some optimization of the multiplication factors we had a well performing rule based agent.

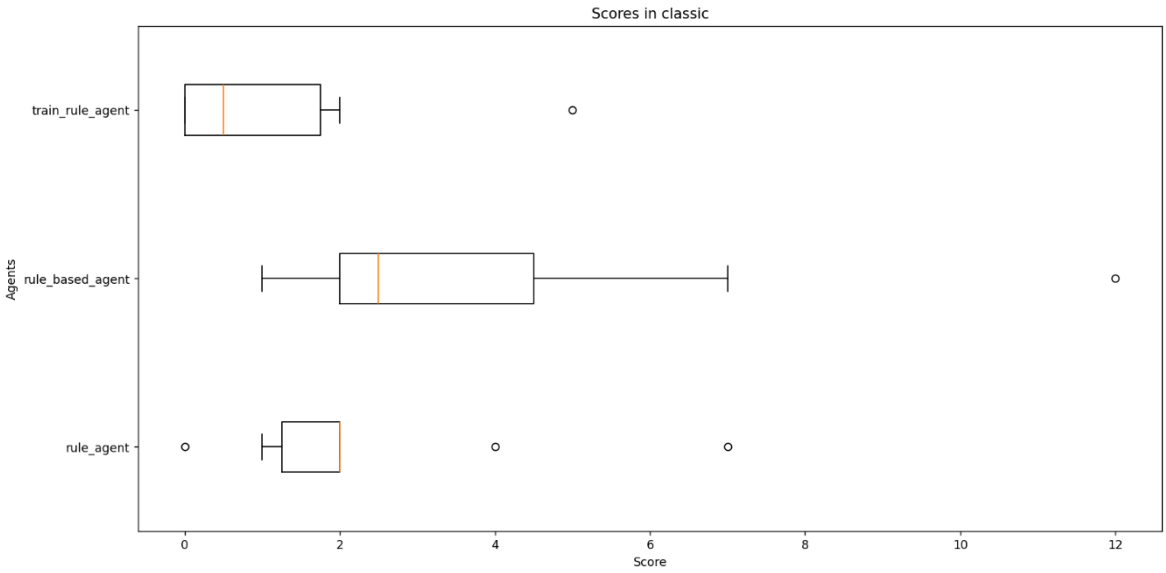
Since our features weren't designed to create a optimal performing rule based agent, rather than a well performing trained model the performance of our rule_agent was compared to the given rule_based agent rather mediocre. But there were also games were our rule_agent won the game.

For debugging the training function of the DQL based agents (we had already abandoned the PPO agents) we used the binary output action vector of the rule_agent. We gave a simple single-branch neural net the output of the rule_agent as input tensor, initialized the net with random weights and biases and trained it with our already implemented Deep Q-learning training function in the established training framework, adapted for the simplified training situation (single step updates, higher learning rate of 0.1, regular ReLu due to binary input). A positive reward of +100 was given if the right/non-zero action was taken, a negative reward of -100 was given if another action was taken. In other words we used highly abstract and very pre-processed features to train our agent to mimic/translate correctly the rule_agent of ours.

That way we confirmed that our DQL training function should be able to train a model on the given features succesfully and at the same time created our final train_rule_agent. Initially this agent was only supposed to stay a form of debugging. But after facing the frustrating results of our other agents and heard from other groups that they as well used similar highly preprocessed features, we decided

to finally submit our train_rule_agent, since it was the only model successfully fulfilling all desired tasks (coin-collection and hunting of other agents).

Below is the performance of our train_rule_agent compared to the rule_based_agent and rule_agent in 10 games in the “coin-heaven”- and “classic”-scenario displayed:



As seen above, our model with abstract features performs quite well in the coin-heaven scenario, but has its problems is the classic-scenario with crates. In both cases, it performs worse than the rule_agent it is based on, which might be due to the simplified training environment. Since it was never supposed to be the agent to be submitted rather than a form for debugging. Putting more focus on the training process of this agent probably would have led to a similar performance as the rule_agent.

8. Conclusion (Jiufeng. Li)

By training, comparing, optimizing the parameters multiple models, we deepened our understanding of reinforcement learning. Q-learning is a popular reinforcement learning (RL) algorithm, which is a model-free, non-policy RL algorithm designed to find the optimal policy for an agent to maximize its cumulative rewards in a Markov Decision Process (MDP) or similar environment. As a model-free algorithm, it does not require prior knowledge of the dynamics or transition probabilities of the environment, but learns directly from interaction with the environment. However, the basic form of Q-learning relies on a tabular representation of Q-values, which becomes impractical for problems with a continuous state space, and in our bomberman game, the state space is relatively large, so it appears to be less appropriate for a representation in a q-table, even though it is only a discrete space.

At the mean time, we tried using the Proximal Policy Optimization method, in contrast to Proximal Policy Optimization (PPO) which learns the optimal policy or strategy for actions in different states directly without estimating the action values, thus exploring the new method more naturally. However, it was found through model training that this strategy does not show better performance in bomberman game.

Thereafter, we tried using neural networks, the key theoretical basis of deep Q learning is the use of deep neural networks to approximate Q-functions. The neural network takes the state as input and

outputs the Q value for each action. At the same time, we compared the difference in performance between a simple network and a convolutional network. After pairing the experimental results, we concluded that simple_net_q performs better because it can handle a larger number of features with lower dimensions, so the model performs better.

9. Literature

Bengio, Y., Louradour, J., Collobert, R., & Weston, J. (2009). Curriculum Learning. *Proceedings of the 26th Annual International Conference on Machine Learning (ICML)*.

Fitzgibbon, R. A. (2001). Robust Registration of 2D and 3D Point Sets. *British Machine Vision Conference (BMVC).* [Online] Available at: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/fitzgibbon-bmvc-2001.pdf.

Kingma, D. P., & Ba, J. (2014). Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*.

Mnih, V., Kavukcuoglu, K., Silver, D., et al. (2013). Playing Atari with Deep Reinforcement Learning. *NIPS (Neural Information Processing Systems)*. Retrieved from https://arxiv.org/abs/1312.5602.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Petersen, S. (2015). Human-level control through deep reinforcement learning. *Nature, 518*(7540).

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347*.