*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 3](#_Toc146967123)

[2. Introduction (J. Li) 3](#_Toc146967124)

[3. Methods (J. Li) 3](#_Toc146967125)

[4. Feature Engineering (C. Zehender) 3](#_Toc146967126)

[5. Training framework (C. Zehender) 4](#_Toc146967127)

[5.1 Performance tracking (C. Zehender) 4](#_Toc146967128)

[5.2 Curriculum learning (C. Zehender) 5](#_Toc146967129)

[5.3 Exploration vs. Exploitation (J. Li) 6](#_Toc146967130)

[5.4 Loss calculation (C. Zehender) 6](#_Toc146967131)

[5.5 Single step vs. batch updates (C. Zehender) 7](#_Toc146967132)

[5.6 Memory replay (C. Zehender) 7](#_Toc146967133)

[5.7 Reward shaping (C. Zehender) 7](#_Toc146967134)

[5.7.1 Reward normalization (C. Zehender) 7](#_Toc146967135)

[5.7.2 Reward-based imitation learning vs. Self-customized reward functions (C. Zehender) 8](#_Toc146967136)

[6. Network design (C. Zehender) 11](#_Toc146967137)

[6.1 NaN Handling and Handling of zeros (C. Zehender) 11](#_Toc146967138)

[6.2 Single-branch network vs. Multi-branch network (C. Zehender) 11](#_Toc146967139)

[7. Experiments and Results (C. Zehender) 13](#_Toc146967140)

[7.1 Deep Q-Learning (C. Zehender) 13](#_Toc146967141)

[7.1.1 Hyperparameter optimization (C. Zehender) 14](#_Toc146967142)

[7.2 Proximal Policy Optimization (C. Zehender) 16](#_Toc146967143)

[7.2.1 Hyperparameter optimization (C. Zehender) 16](#_Toc146967144)

[7.2.2 Results (C. Zehender) 16](#_Toc146967145)

[7.3 Debugging (C. Zehender) 16](#_Toc146967146)

[8. Conclusion (J. Li) 17](#_Toc146967147)

[9. Literature 17](#_Toc146967148)

# 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 (J. Li)

# 3. Methods (J. Li)

# 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.

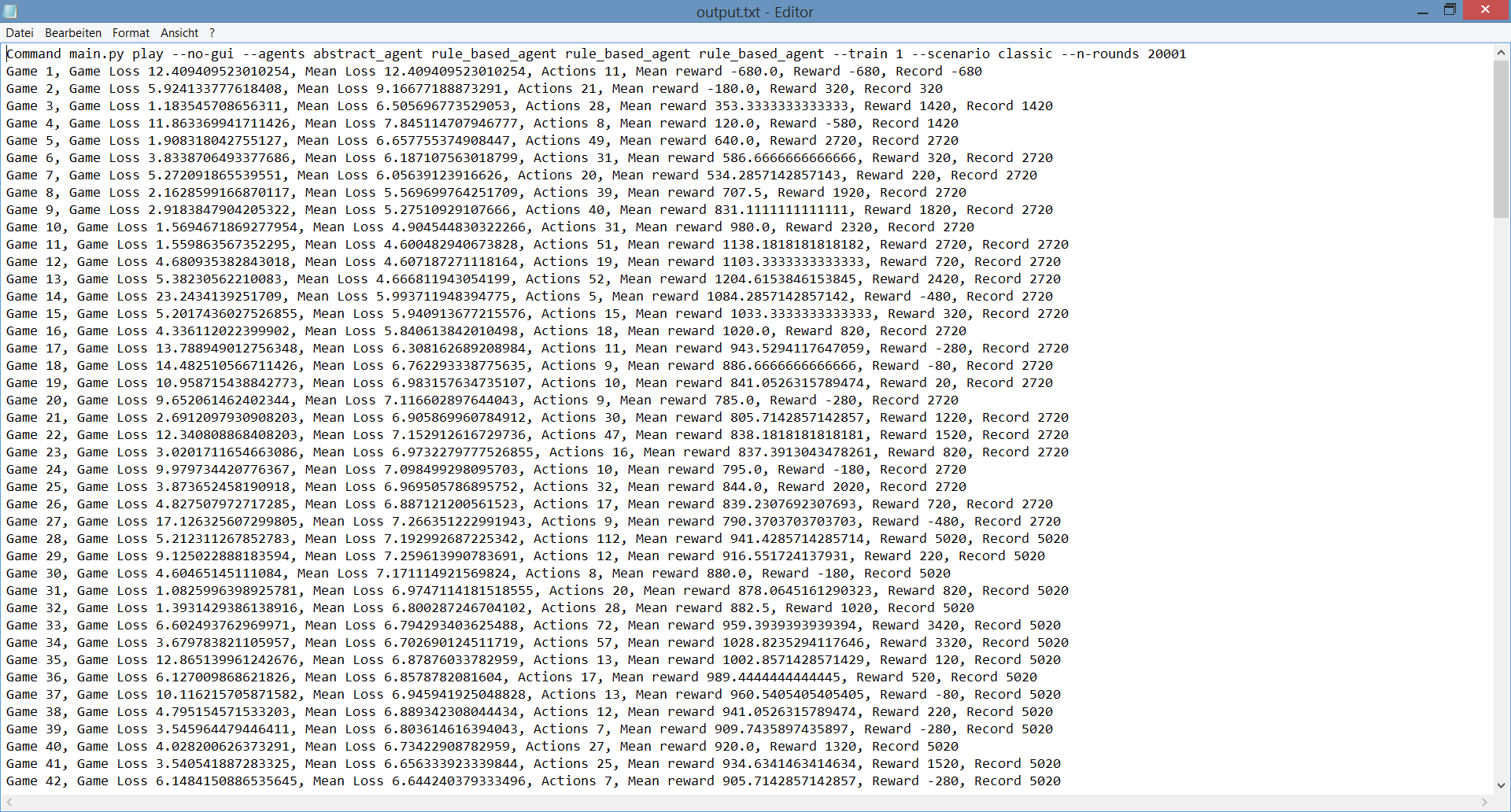
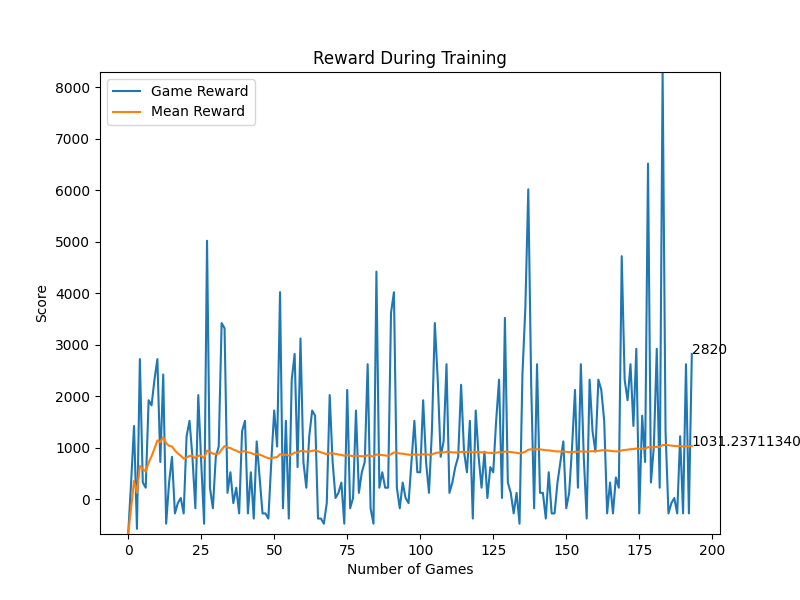
After starting to create a multi-branch network, we created a list of input tensors, with each tensor of that list supposed to go into its own branch. Those tensors need to be the same size. Therefore, we created another feature. It calculates, without taking walls, stones or crates into account, the 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 (C. Zehender)

## 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.



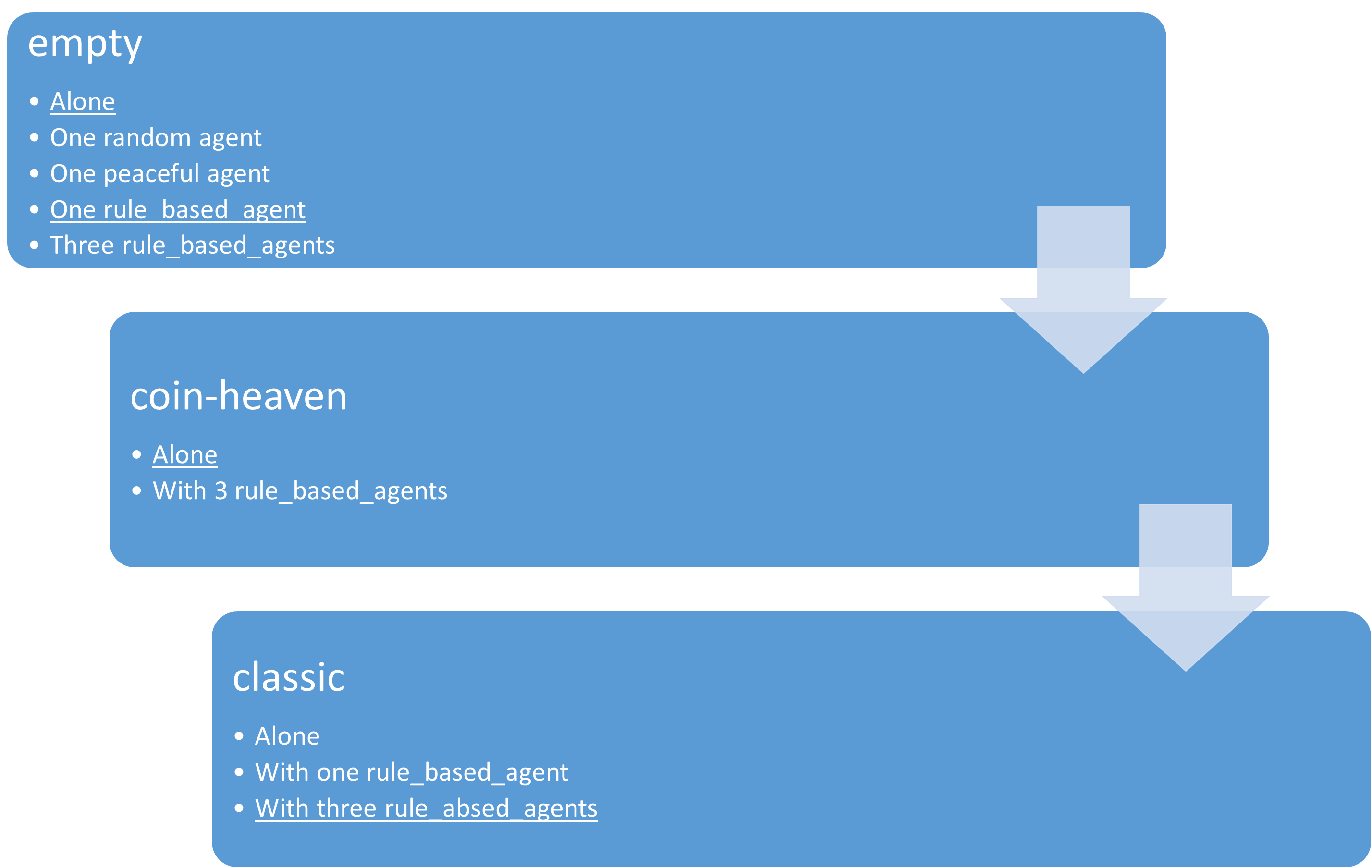
**Exemplary output for performance tracking:**

## 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 (J. Li)

## 5.4 Loss calculation (C. Zehender)

For 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 Q-learning 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:

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 (C. Zehender)

### 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:

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 Q-learning, 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 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.

Reward table:

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



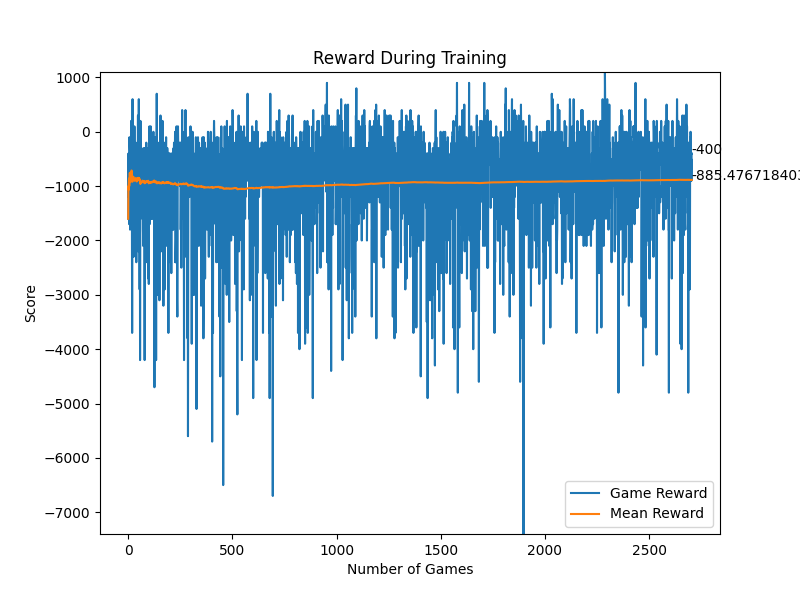
One special reward function, the train\_coin\_collected-function should be described more in detail:

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:

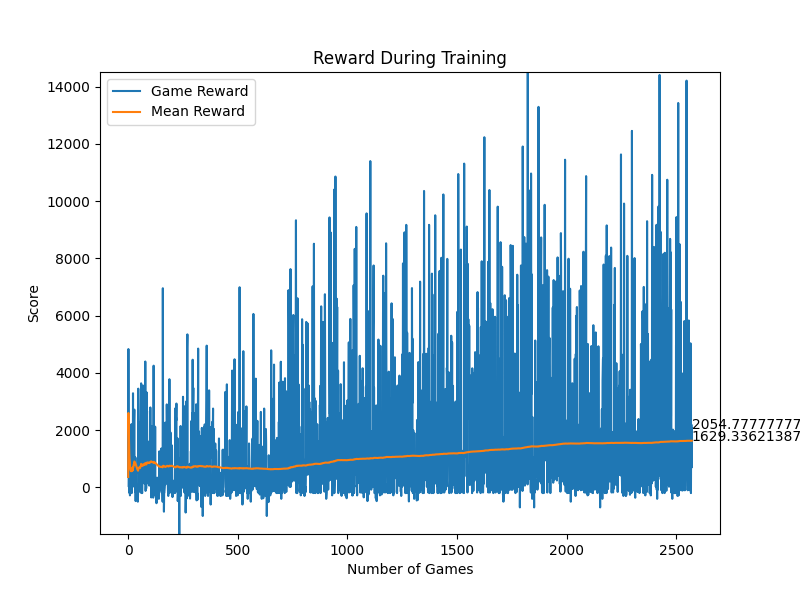
With . The scaling in the formula ensures a reasonable decrease in the reward while collecting a coin still gives at least a reward of 24 for all actions that brought the agent closer to the collected coin since the last one was collected without jeopardizing itself. By applying this formula the agent 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:

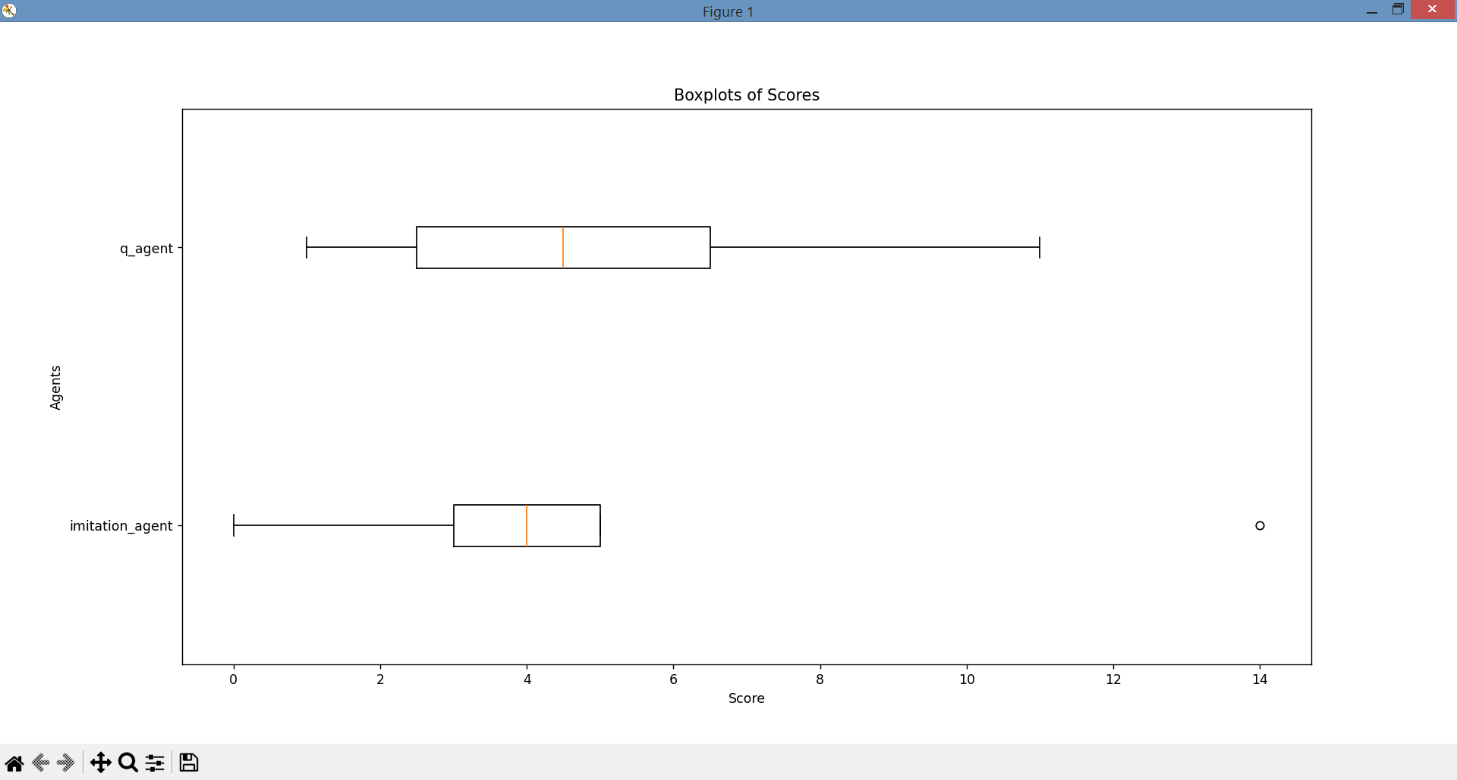


This diagram compares the the rewards per game during training of the imitation\_agent (imitation learning) and q\_agent (self-defined rewards) trained on batch updates and a multi-branch net with a random\_vec of 0.25 under the command main.py play --no-gui --agents simple\_net\_q rule\_based\_agent --train 1 --scenario coin-heaven --n-rounds 20001.

imitation\_agent)



q\_agent)

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 reward is increasing 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:

As seen at above boxplots, both agents perform after more than 2500 played games very similar according to their score, which could also be due to their not so good average performance of collecting 4 coins. When observing the patters and reactions of the agents visually the agent based on imitation learning might be a little bit 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 the reward-based imitation learning.

# 6. Network design (C. Zehender)

## 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.

1. 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.

1. 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.

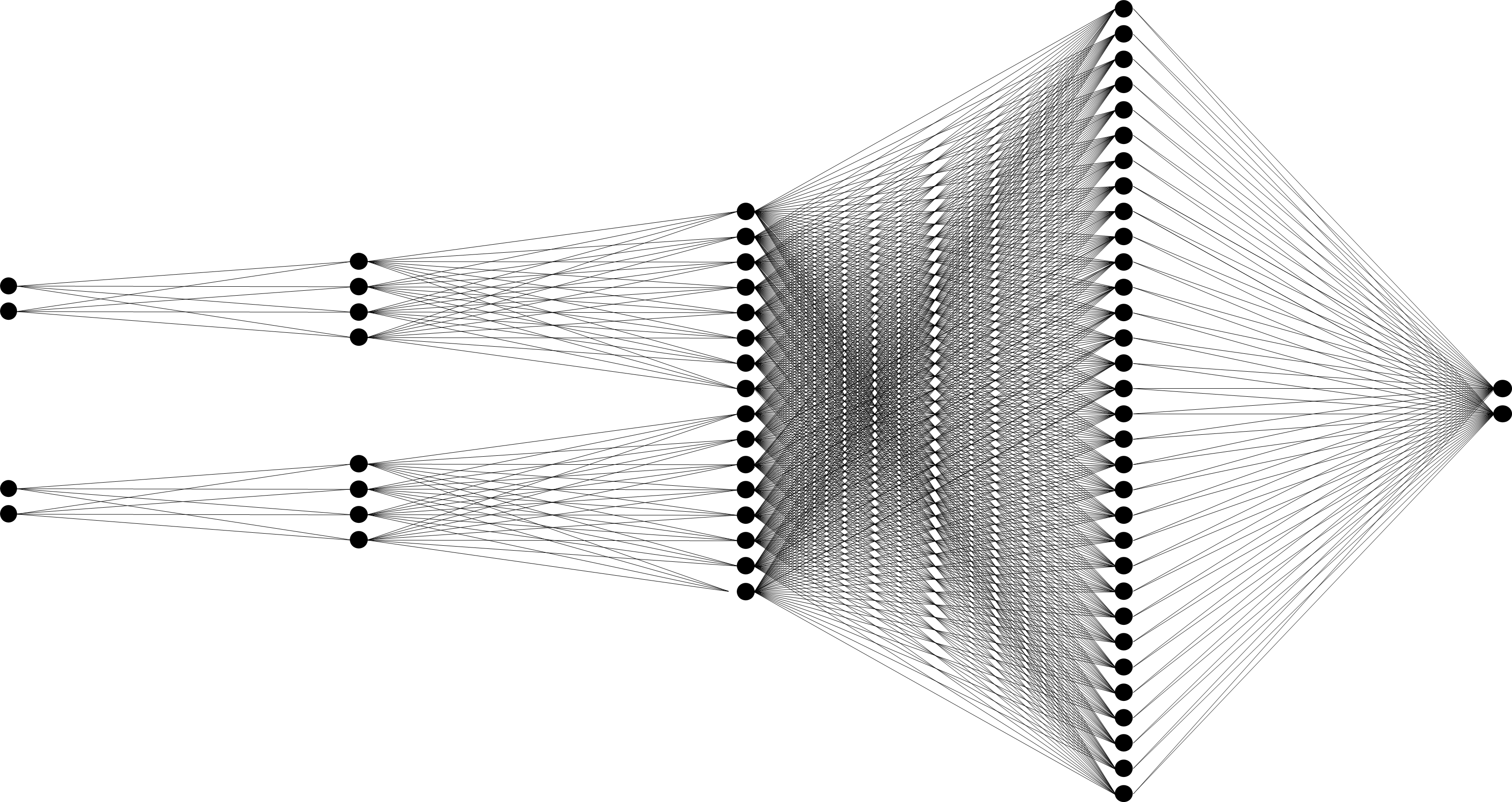
## 6.2 Single-branch network vs. Multi-branch network (C. Zehender)

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](https://www.dict.cc/?s=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:

*Input*

*1st hidden layer*

*Stacked branch output*

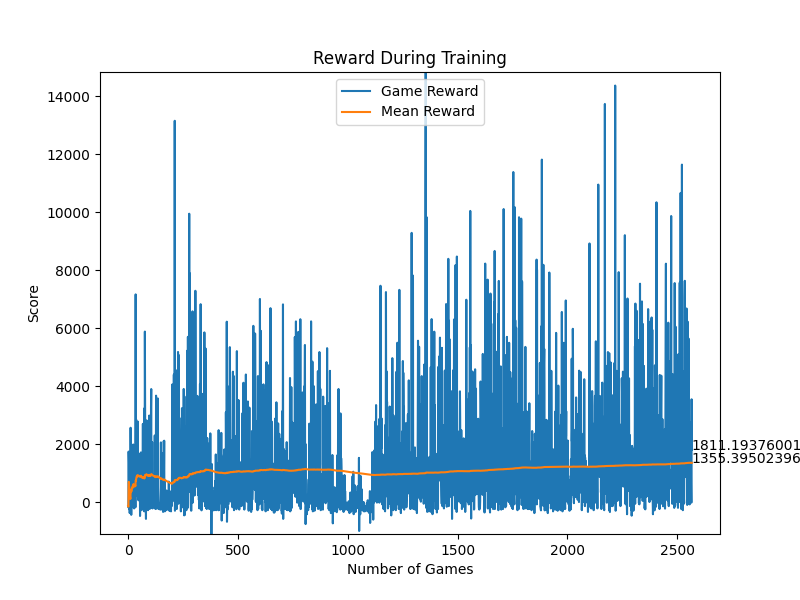
*Final hidden layer*

*Output*

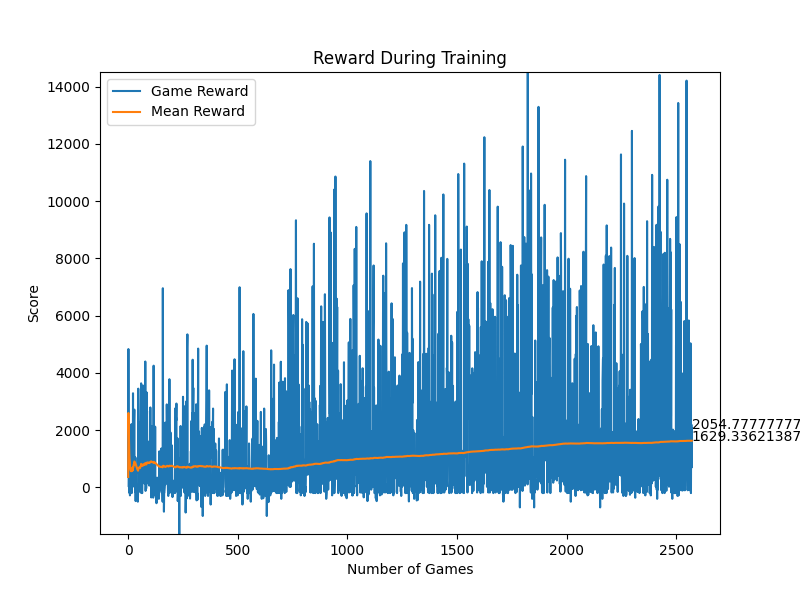
Schema of a multi-branch network. For better visualization the input sizes and layer sizes were reduced.

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.



simple\_net\_q)



q\_agent)

As seen in the above diagrams, the single-branch net is just slightly 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 simple enough that the model does not benefit that much from the complexity of multi-branch networks. So we concluded that the task could also be effectively solved by processing all input features together, but continued with the multi-branch net due to its better performance.

# 7. Experiments and Results (C. Zehender)

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 is as follows:

Where:

is the Q-value for state and action .

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

is the immediate reward received after taking action in state .

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

is the maximum Q-value for the next state over all possible actions . (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 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 100th 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 10th 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.

Scaling factor for normalization: Most of the time, a scaling factor of 1 was applied. If the loss of the individual steps showed low variance within our txt.-output file, we decreased it to 0.1, 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 > α > 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.

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, own rewards) and rule\_based\_agent in the coin-heaven scenario and the classic scenario with three other rule\_based\_agents:

## 7.2 Proximal Policy Optimization (C. Zehender)

Since the results for Q-Learning weren’t that satisfying 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 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. It aims to ensure that the policy update does not deviate too far from the current policy to maintain stability during training. The use of a surrogate objective and a clipping mechanism helps create a balance between exploration and exploitation, making PPO suitable for our task.

### 7.2.1 Hyperparameter optimization (C. Zehender)

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

### 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 imitation\_agent (DQL, imitation-based), q\_agent (DQL, own rewards) and rule\_based\_agent in the coin-heaven scenario and the classic scenario with three other rule\_based\_agents:

## 7.3 Debugging (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\_feature with our self-customized multipliers 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 well 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. The average score of our rule\_agent was , while the average score of the rule\_based\_agent was .

For debugging the training function of the Q-Learning 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 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 Q-Learning 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 the “classic”-scenario displayed:

# 8. Conclusion (J. Li)

# 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).