*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“ in a simplified version.

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](#_Toc146773485)

[2. Introduction 3](#_Toc146773486)

[3. Methods 3](#_Toc146773487)

[4. Feature Engineering 3](#_Toc146773488)

[5. Training framework 3](#_Toc146773489)

[5.1 Performance tracking 3](#_Toc146773490)

[5.2 Curriculum learning 3](#_Toc146773491)

[5.3 Exploration vs. Exploitation 4](#_Toc146773492)

[5.4 Loss calculation 5](#_Toc146773493)

[5.5 Single step vs. batch updates 5](#_Toc146773494)

[5.6 Memory replay 6](#_Toc146773495)

[5.7 Reward shaping 6](#_Toc146773496)

[5.7.1 Reward normalization 6](#_Toc146773497)

[5.7.2 Imitation learning 6](#_Toc146773498)

[5.7.3 Hand crafted reward functions 6](#_Toc146773499)

[6. Network design 7](#_Toc146773500)

[6.1 NaN Handling and Handling of zeros 7](#_Toc146773501)

[6.2 Single-branch network vs. Multi-branch network 7](#_Toc146773502)

[7. Experiments and Results 9](#_Toc146773503)

[7.1 Q-Learning 9](#_Toc146773504)

[7.1.1 Hyperparameter optimization 9](#_Toc146773505)

[7.1.2 Results 9](#_Toc146773506)

[7.2 Proximal Policy Optimization 9](#_Toc146773507)

[7.2.1 Hyperparameter optimization 9](#_Toc146773508)

[7.2.2 Results 9](#_Toc146773509)

[7.3 Debugging 9](#_Toc146773510)

[7.3.1 Feature debugging 9](#_Toc146773511)

[7.3.2 Training debugging 9](#_Toc146773512)

[8. Conclusion 10](#_Toc146773513)

# 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 under the title of said subsections.

# 2. Introduction

# 3. Methods

# 4. Feature Engineering

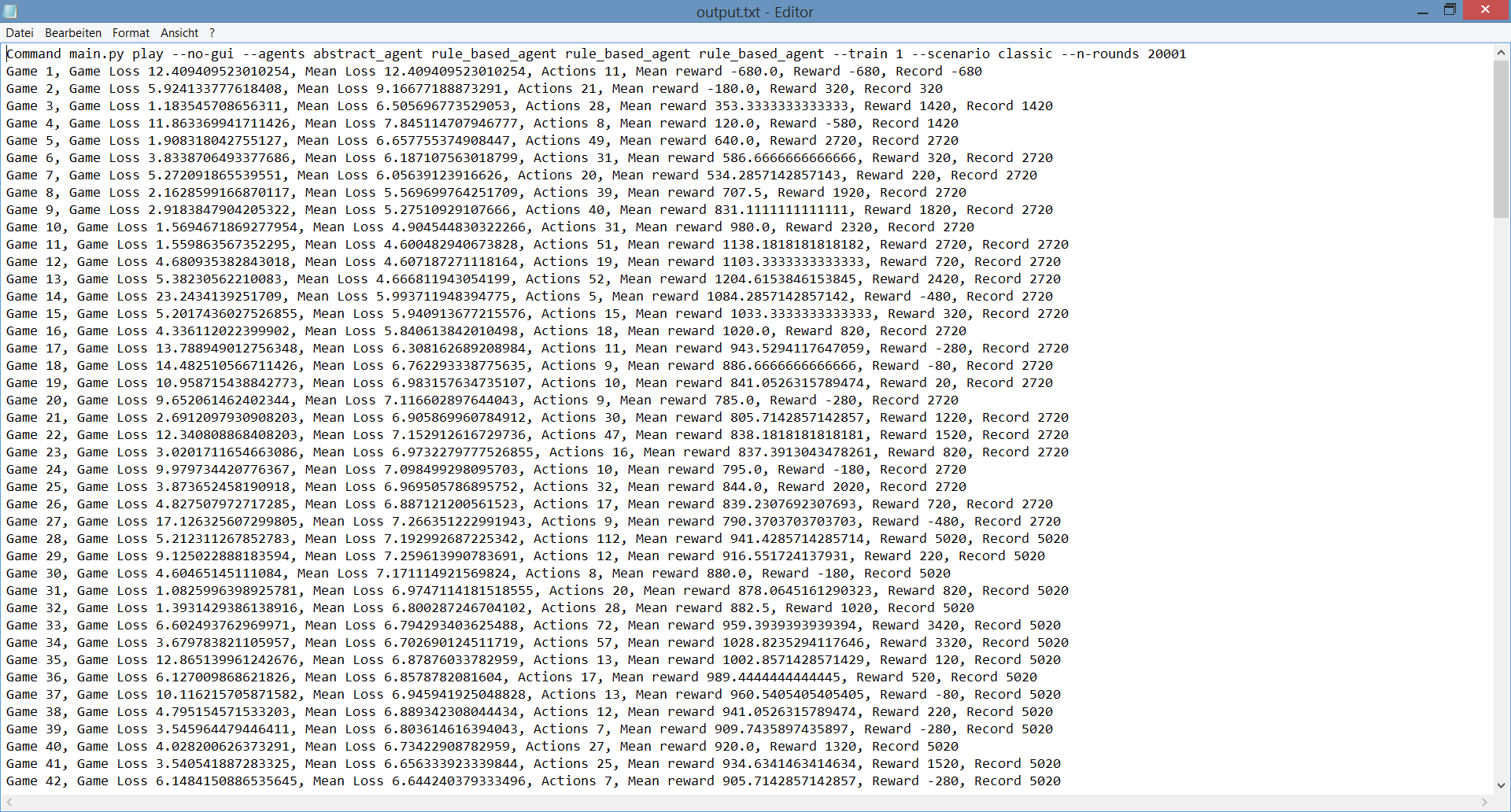
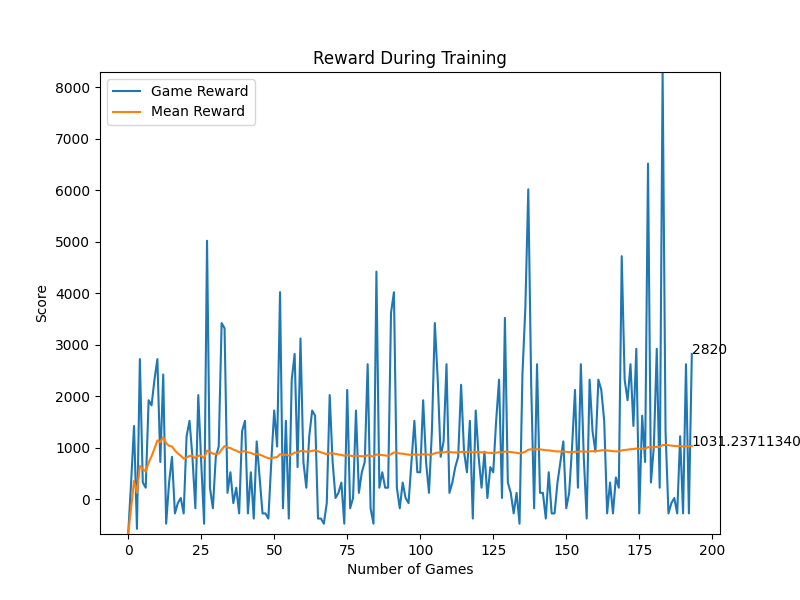
As in the previous section described the huge amount of possible states makes it necessary to condense the most important information about the field into an overseeable number of features.

# 5. Training framework

## 5.1 Performance tracking

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 behavior 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 mean per game and the average loss and mean 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.



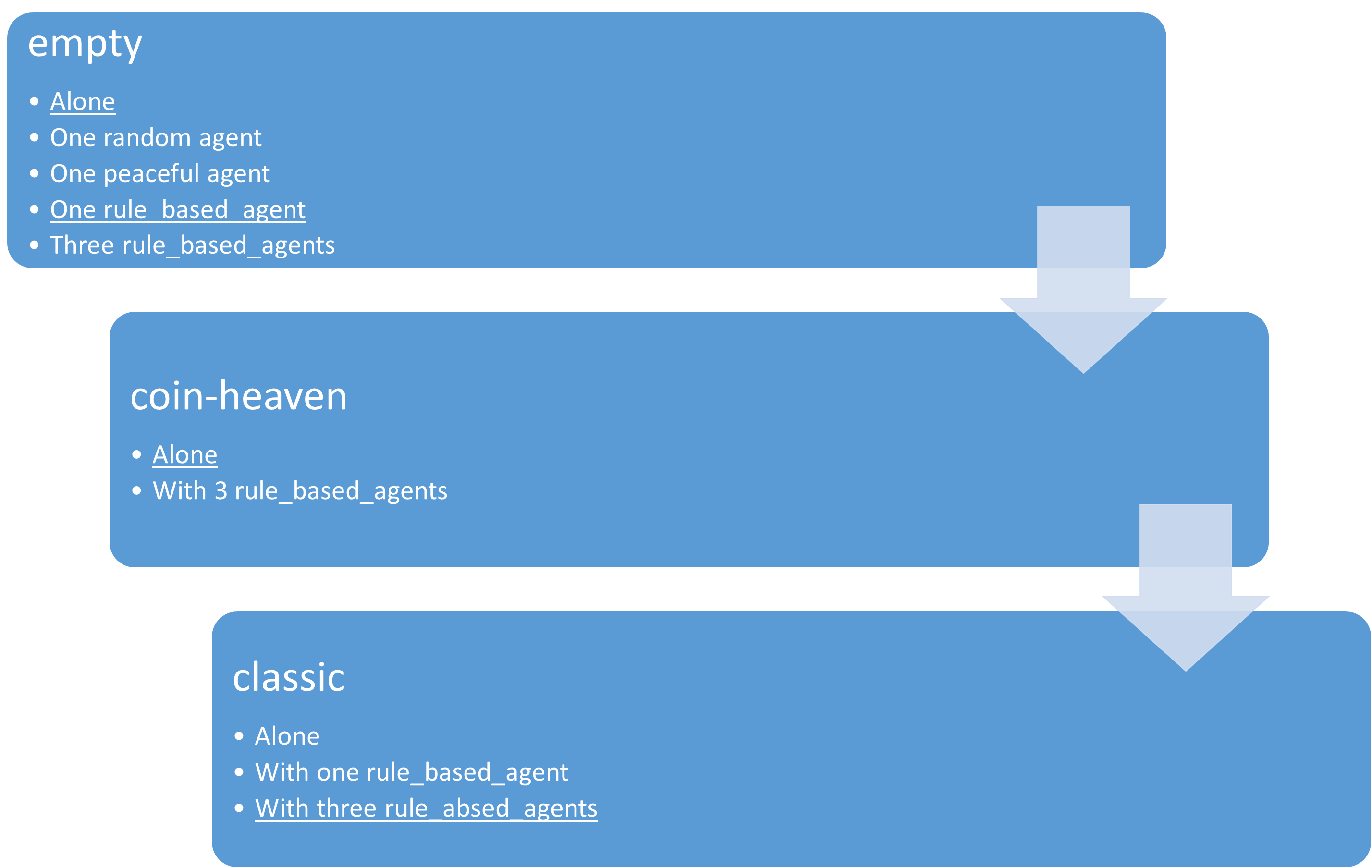
**Exemplary output for performance tracking:**

## 5.2 Curriculum learning

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:

**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 simple 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 employed our later described ε–greedy strategy according to our self-defined rewards (bzw. later then imitation learning). Thereafter, we introduced another agent.

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

## 5.4 Loss calculation

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 through our ε–greedy strategy, 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.

## 5.5 Single step vs. batch updates

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

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." DQN was one of the pioneering algorithms that successfully combined deep neural networks with 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.

## 5.7 Reward shaping

### 5.7.1 Reward normalization

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 excessively high rewards. 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. 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 Imitation learning

### 5.7.3 Hand crafted reward functions

Going trough them in detail would go beyond the scope of this report. But shortly.

# 6. Network design

## 6.1 NaN Handling and Handling of zeros

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 are using 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.

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.

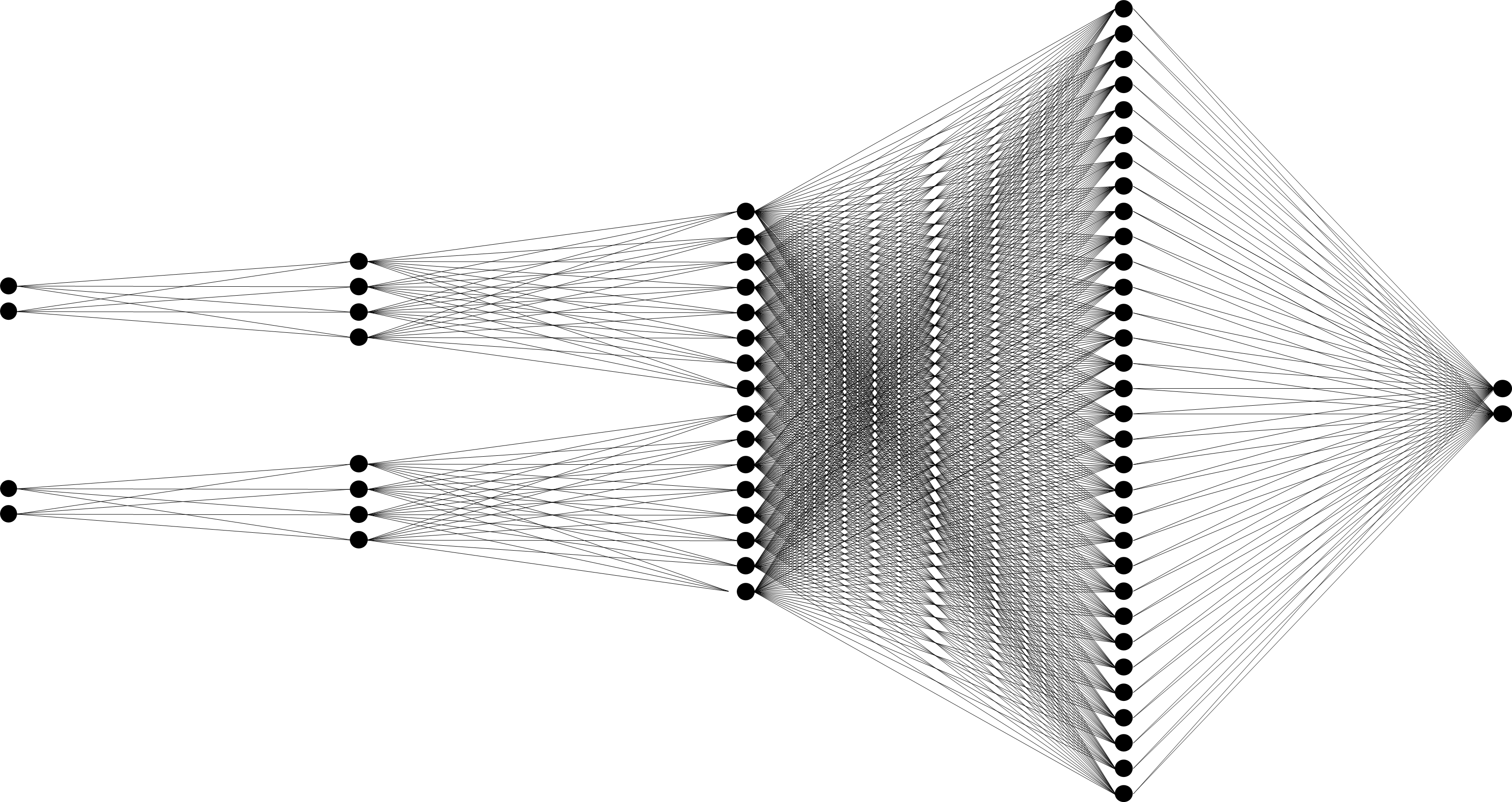
## 6.2 Single-branch network vs. Multi-branch network

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. 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 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 task and respectively 4 branches:

* Environment Awareness Branch: Gets the [reasonableness](https://www.dict.cc/?s=reasonableness) of all 6 actions as binary input.
* Coin collection branch: Gets the relative coordinates of the nearest crate and coin, as well as the center 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 center 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 center 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 2 hidden linear layers 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 a Multi-branch network:

*Input*

*1st hidden layer*

*Stacked branch output*

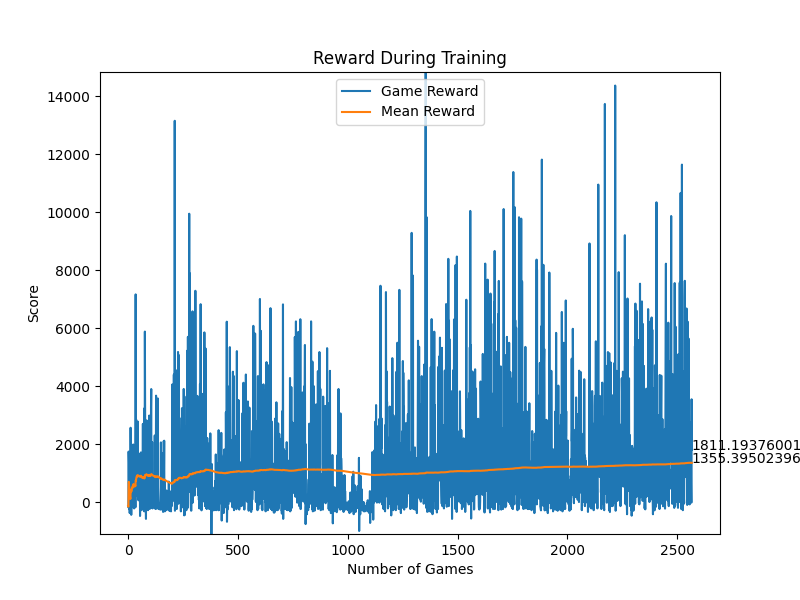
*Final hidden layer*

*Output*

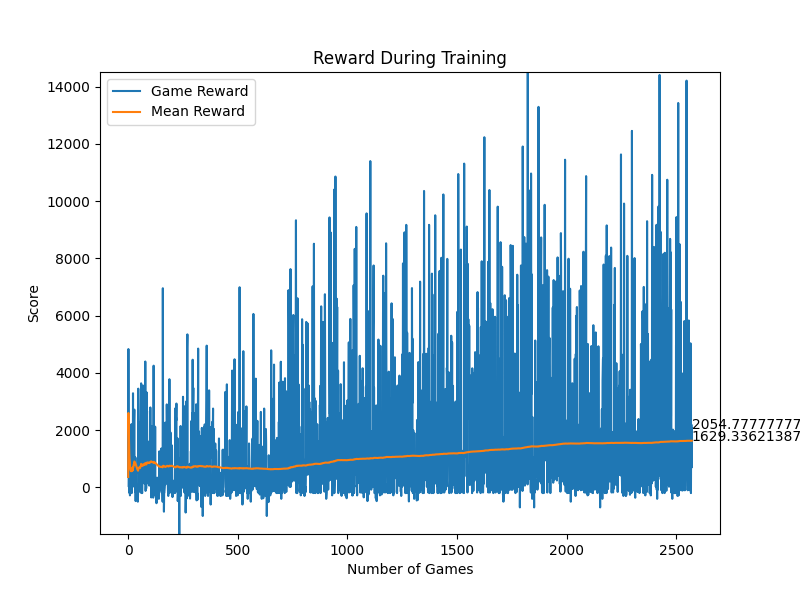
Schema of a multi-branch network. For better visualization the input sizes and layer sizes were reduced.

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

## 7.1 Q-Learning

### 7.1.1 Hyperparameter optimization

### 7.1.2 Results

## 7.2 Proximal Policy Optimization

### 7.2.1 Hyperparameter optimization

### 7.2.2 Results

## 7.3 Debugging

### 7.3.1 Feature debugging

### 7.3.2 Training debugging

# 8. Conclusion