Universität Heidelberg

Machine Learning Essentials(2024)

Reinforcement Learning for Bomberman

Authors
Suryansh Chaturvedi
Felix Exner
Dorina Ismaili

Supervisor
Ullrich Köthe

Github repository:
https://github.com/suryansh98/Bomberman-MLE-2024-Mlbot

30/09/2024

Contents

1 Introduction

1.1 Overview

Dorina Ismaili

Bomberman, originally developed as a maze strategy game in Japan, revolves around deftly maneuvering through maze environments, eliminating adversaries, and collecting coins to accumulate points.
The core dynamics of the game naturally direct us to reinforcement learning, an advanced tool used to master the intricacies of the game. Its multiplayer dimension has had a huge interest within the machine learning community, prompting exploration of sophisticated reinforcement learning methodologies, which in turn have catalyzed the development of novel, cutting-edge algorithms for excelling in this dynamic competitive arena.
In this project, we will design two reinforcement learning agents to compete in the tournament. Their goal is to win the game, by using learned strategies to navigate the maze, evade opponents, collect coins, and utilize power-ups effectively. This implementation will offer valuable insights into the agents' decision-making processes and the relative efficacy of various reinforcement learning techniques.
Following the lecture, we will delve into similar algorithms for developing our reinforcement learning agent. Additionally, various methodologies will be applied, like feature engineering, q-learning, deep q-learning, etc. each demonstrating the agent's capacity to adapt to complex environments. Each approach will be evaluated based on its performance in the maze, making sure the agent can efficiently navigate, collect resources, and compete against opponents. The experiments and their outcomes are critical benchmarks for assessing the success of our agent in the Bomberman game competition.

1.2 Problem Description

Dorina Ismaili

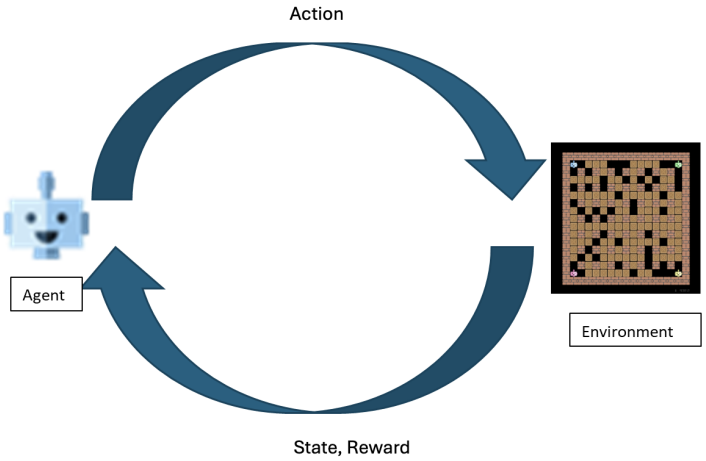
In this scenario, the game is played by four agents operating in discrete time steps, where each agent can perform one of six possible actions: moving left, right, up, or down; placing bombs; or waiting. The strategic combination of these actions allows agents to navigate through the maze. Effective decision-making in response to

both the environment and other agents will be crucial for maximizing performance and achieving victory in the game.

The tasks for the reinforcement learning agent involve:
- Placing bombs strategically (to destroy crates and eliminate enemies)
- Collecting coins
- Avoiding bomb blasts (both the agent's and those of opponents)
- Navigating the grid to maximize long-term survival and reward
- Beating other agents.

We use feature engineering to implement these states of the agent.

Additionally, the maze is composed of walls and crates. By strategically placing bombs, they can destroy crates, revealing hidden coins. Collecting these coins earns players valuable points, encouraging them to explore different areas of the maze while also competing against opponents. The game is placed in a 17x17 grid where each agent stands in the corner of the field. As decided on the environment, in this 289 tiles, 113 are static walls, hence leaving 176 tiles on which the game takes place. There are a total of 9 coins and 132 crates randomly placed. Each game round unfolds over the course of 400 discrete time steps, providing a finite window for agents to execute their strategies and compete for dominance. The agent's performance will be tested on different scenarios like a coin-heaven board; a board with hidden coins on crates, a board with crates and beating other agents.



1.3 Reinforcement Learning for Bomberman

Dorina Ismaili

Reinforcement Learning (RL) techniques are effectively used to tackle the strategic challenges presented by Bomberman, wherein agents learn to make decisions through interaction with the game environment. In Bomberman, the agent learns to navigate in the playing field, decides when and where to place bombs, collect coins and kill opponents to maximize the rewards. We can model problems of this nature in terms of optimal control of Markov Decision Processes, where the goal is to model the decision-making dynamics of the agent as it interacts with its environment.
By modeling the game's fundamental components—state transitions, actions, rewards, and policy learning—the agent can progressively learn and refine its strategies over time. An agent must be able to *sense* the state of the environment and must be able to take *actions* that affect the state. The agent also has a *goal* or goals relating to the state of the environment. The formulation includes these three aspects: sensation, action, and goal.
The interaction between the agent and the environment occurs over a sequence of discrete time steps. At each time step t, the agent receives a representation of the environment's state, $S_t \in S$, and on that basis selects an action, $A_t \in A(S_t)$. One time step later, as a result of its action, the agent receives a numerical reward , $R_{t+1} \in R$, and finds itself in a new state, S_{t+1}. For a more comprehensive explanation of reinforcement learning, we refer to the book by Sutton [**RL**].

1.4 Related Work

Suryansh Chaturvedi

The application of reinforcement learning to Bomberman has been explored in various research papers, showcasing the effectiveness of different RL techniques in this game environment.

Here are some notable examples:

Multi-Agent Deep Reinforcement Learning for Bomberman[Sta16]: This paper investigates the use of Deep Q-Networks (DQNs) and Double DQNs in a multi-agent setting for Bomberman. The agents learn to cooperate and compete against each other, showcasing the potential of deep RL in complex multi-agent scenarios.

Monte Carlo Tree Search for Bomberman[AA22]: This work explores the application of Monte Carlo Tree Search (MCTS) for playing Bomberman. MCTS is a powerful search algorithm that has proven effective in games like Go and Chess. This paper demonstrates how MCTS can be adapted to handle the strategic challenges of Bomberman.

Evolutionary Algorithms for Bomberman[Aut21]: This research investigates the use of evolutionary algorithms (EAs) for evolving effective playing strategies for Bomberman. EAs are population-based optimization algorithms that mimic natural evolution to find solutions. This paper shows that EAs can be used to learn effective strategies in Bomberman.

These are just a few examples of the growing body of research exploring the use of RL for Bomberman. By building upon these prior works, our project aims to contribute to this field by investigating the effectiveness of Q-learning with feature engineering and exploring new features that improve agent performance.

1.5 Project Planning

Dorina Ismaili

The project was structured around specific goals and milestones for each meeting.

1. *Brainstorming and research*: We started by brainstorming with different reinforcement learning algorithms and reviewing literature, while deciding on an optimal algorithm to implement.
2. *Initial setup*: We set up the agent's code and became familiar with the provided

environment.
3. *Model tuning and training*: We adjust and train the more accurate algorithmic models and tune the ML hyperparameters.
4. *Performance testing and Debugging*: We compare the agent's performance and adjust failure cases to improve performance.
5. *Final testing and documentation*: We run final test to evaluate the agent's performance and document our results.

The tasks were distributed accordingly, as presented in the report, and we ensured that we reached the milestones assigned to each team member.

1.6 The Final Agent

Dorina Ismaili

In the end, we have two well trained agents. The implemented algorithms are Q-learning and Deep Q-Network. Given our experiments, we decided to choose the model with highest chance of winning. In our case, the Q-learning agent demonstrated a significant potential and enriched our way to further research in the field. The result we demonstrate have proven to be very promising. Our final agent is built using Q-learning with a significant emphasis on feature engineering. Features are used to represent the game states and given that Q-learning uses a state-action representation, we are able to integrate the reward winning algorithm. This method has shown robust performance and stability in training. Our agent is able to collect coins, place bombs, kill opponents and sometimes win the game. It's performance is tested in the different scenarios and tasks provided on the project like on a coin-heaven board, on a board with crates where you find the hidden coins, blowing up other agents and finally beating the rule_based_agent. In addition, the performance of our agent was improved by tuning the hyperparameters, allowing us to exploit and explore. Our main goal was to win the game.

1.7 Repository

2 Methods

In this section, we outline various methods we have utilized for training agents through reinforcement learning algorithms. These methods can be categorized into value-based, policy-based, and model-based approaches.

2.1 Markov Decision Process

Dorina Ismaili

The Markov property for a reinforcement learning problem states that the environments response at time $t+1$ depends only on the current state S_t and action A_t representation at time t, rather then on prior states or actions [SB18]. Mathematically it is represented as

$$p(s', r|s, a) = P(R_{t+1} = r, R_{t+1} = s'|S_t, A_t),$$

where all the dynamics are encapsulated in the current state.

A Markov decision process is a tuple (S, A, T, R, γ) consisting of:
- S: State space,
- A: Action space,
- T: Transition function $P(s'|s, a)$; the probability of transitioning from s to s' under action a
- R: Reward function $R(s, a, s')$
- γ: discount factor; consider future rewards.

Given the described dynamics, we can compute all the interaction with the environment such as expected reward for state-actions pair, the state-transition probabilities, etc.

The solution to this decision problem is function $\pi : S \to A$, called the policy,

$$\pi(s) = arg\max_{a \in A} \mathbb{E}[R],$$

which maximized the expected cumulative reward and R describes the sum of re-

wards discounted by a factor $\gamma \in [0, 1)$ expressed as

$$R = \sum_{t=0}^{\infty} \gamma^t r_t = r_0 + \gamma r_1 + \gamma^2 r_2 + \dots$$

The value of a state s under a policy π, denoted by $v_\pi(s)$ is the state value function such that

$$v_\pi(s) = \mathbb{E}_\pi[R|S_t = S].$$

In the field of reinforcement learning, several algorithms are responsible for training an optimal policy. Furthermore, these processes highlight the exploration-exploitation trade-off in RL.

2.2 Q-learning

Dorina Ismaili

Q-learning is a model-free, value-based, off-policy algorithm aimed at identifying the optimal sequence of actions based on the agent's current state. The "Q" refers to quality, indicating how crucial a particular action is in terms of maximizing future rewards. In value-based approaches, the algorithm trains a value function to evaluate which state holds greater value, guiding the agent's actions accordingly. Conversely, policy-based methods directly train the policy itself, enabling the agent to learn which action to take in a specific state [WD92].

In our Bomberman game, we employ Q-learning method as a reinforcement-learning technique. We introduce briefly the components of the Q-learning algorithm.
The Q-function uses Bellman equation and takes states and actions as input:

$$Q(s_t, a_t) = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | s_t, a_t]$$

The agent utilizes a Q-table to determine the optimal action based on the received rewards. The Q-table is a data structure initialized according to the number of states and actions available.

The agent selects an action and updates the Q-table based on the outcome of that action. The Epsilon-Greedy Strategy is a straightforward approach that balances exploration and exploitation. The epsilon value represents the probability of choosing to explore new actions, while exploitation occurs when there is a lower likelihood

of exploration, favoring known actions with higher expected rewards [Tri21].

We update the Q-function by using estimated Q-values, learning rate and temporal difference error.

The update rule is repeated many times until the Q-table is updated and Q-value is maximized. It is expressed as:

$$Q^{\text{new}}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max Q(s_{t+1}, a_t) - Q(s_t, a_t)], \tag{1}$$

where t is the current time step in the episode.
- a_t is the action during the t timestep,
- s_t is the state during the t timestep,
- r_t is the reward returned for the action,
- s_{t+1} is the subsequent state after action a_t,
- α is the learning rate, determines to what extent new information overwrites the old information
- γ is the discount factor $\gamma \in [0, 1)$, determining the importance of future rewards.

The Q-learning algorithm is a highly effective method for an agent to learn and understand how the environment operates. We use the numpy library to implement the algorithm. Given the large memory of actions and states, it is sometimes hard to implement the working algorithm. In this cases, when we need to take care of the memory we introduce another algorithm wich can approximate this Q-values. In most of the cases, the Deep Q-Network algorithm is used.

2.2.1 Q-Table

Dorina Ismaili

In order to implement the Q-learning algorithm, we initialize the Q-table. The Q-table is a data structure that stores the rewards linked to the best actions for each state within the environment. Given its construction, it guides the agent to choose actions that lead to most reward. As the agent continuous to interact with the environment, the Q-table is continuously updated to reflect the agent's growing knowledge, allowing for more informed and strategic decision-making. The table is constructed using features and the six actions of the agent. In most of the cases, the agent neglects some features and the features are picked from a similar state. The

values are the values of the Q-function for the six actions. The Q-table is updated iteratively.

2.2.2 Exploitation vs Exploration

Dorina Ismaili

In reinforcement learning, we are often introduced with a trade off involving exploration and exploitation. A simple action to take is repeat a decision known to be successful meaning to exploit or choose to explore meaning to make an unknown decision in hope of increasing the reward. There are various methods of exploration techniques, which will be shown depending on the hyperparameters we choose [KDW18]. In addition, we tried different exploration techniques which evolve over the training of our values. Achieving a perfect balance between both techniques is important in achieving a successive learning of our algorithm. Exploitation is a strategy that leverages existing knowledge to make decisions aimed at maximizing the expected reward. In our implementation, we need to take care of this parameter, which decides on the success of the learning of our method. In this scenario, following [Kow+22], we introduce two different exploration techniques.

ϵ-Greedy Exploration: is one of the most used methods that trades off exploration with exploitation. It uses the parameter ϵ to decide on the amount of exploration and exploitation, where for epsilon equal to 1 we have only exploration. The action with the highest Q-value is selected with a probability of $1 - \epsilon$ and a random action is chosen otherwise. A major drawback in the method is the randomness of the exploration actions, hence a second best action is chosen as the worse value.

Max-Boltzmann: is a method which assigns a probability value to all action starting from best to worse action, hence improving the drawback in the ϵ-Greedy algorithm. The probabilities are assigned using a Boltzmann distribution function, where the probability of assigning action a in state s is:

$$\pi(s, a) = \frac{e^{Q(s,a)/T}}{\sum_i^{|A|} e^{Q(s,a_i)/T}},$$

where $|A|$ is the amount of possible actions and the temperature T, is a control parameter. A high temperature (T) leads to increased exploration. Mixing the two methods together, leads to a better exploration behavior.

2.3 Hyperparameter Tuning

Suryansh Chaturvedi

Hyperparameter tuning was an iterative and crucial process in optimizing the performance of our Q-learning agent. We systematically explored different combinations of hyperparameters to find the settings that yielded the best results in terms of our chosen evaluation metrics (discussed in Section 3.3). Here's a breakdown of the tuning process:

2.3.1 Hyperparameters and Ranges

Suryansh Chaturvedi

We focused on tuning the following hyperparameters, based on their significant influence on the Q-learning algorithm:

Hyperparameter	Description	Range Explored	Final Value
Learning Rate (α)	Controls the step size during Q-value updates.	0.01, 0.05, 0.1, 0.2, 0.5	0.1
Discount Factor (γ)	Determines the importance of future rewards.	0.8, 0.9, 0.95, 0.99	0.95
Initial Epsilon (ϵ)	Initial exploration rate.	0.8, 1.0	1.0
Epsilon Decay	Controls the decrease in exploration over time.	0.0001, 0.001, 0.0005	0.0001

2.3.2 Tuning Method

Suryansh Chaturvedi

Manual Tuning: We began with manual tuning, starting with commonly used default values for Q-learning. We observed the agent's performance in the training environment and made adjustments to the hyperparameters based on its behavior. For instance, if the agent was learning too slowly, we increased the learning rate. If the agent was exhibiting unstable behavior, we decreased the learning rate or the discount factor.

Grid Search: To find a more optimal set of hyperparameters, we implemented a grid search. This involved systematically exploring all possible combinations of hyperparameters within the specified ranges. For each combination, we trained the

agent for a fixed number of rounds and evaluated its performance on a validation set of game scenarios.

2.3.3 Selection Criteria

Suryansh Chaturvedi

Win Rate and Average Score: We prioritized hyperparameter combinations that resulted in a higher win rate and a higher average score on the validation set.
Stability: We also considered the stability of the learning process. Hyperparameter settings that led to consistent and smooth improvements in performance were preferred over those that caused erratic fluctuations.
Through this comprehensive hyperparameter tuning process, we were able to significantly enhance our agent's learning efficiency and overall performance in the Bomberman game. The final hyperparameter values presented in the table above represent the best balance we achieved between learning speed, stability, and overall game performance.

2.3.4 Experience Replay

Felix Exner

Experience Replay is a widely used technique in machine learning processes to improve the learning efficiency and stability of the training process. The basic idea is to store the agent's experiences during gameplay in a replay buffer, and then randomly sample from this buffer to train the model. As experience one refers to a tuple (`s, r, a, s'`) consisting of state transitions including the current state `s`, the chosen action `a`, resulting reward `r` and the subsequent state `s`'. For implementing a DQN it is useful to add one more information in the tuple acting as a flag if the tuple resembles a terminate state. This is useful as in this case the loss function of the network slightly differs following the Bellman-equation.

Using experience replay removes correlation between elements within a sample due to randomness of sampling, thus leading to a more stable learning. Further, it leads to more efficient data usage as it allows reusing the stored data.

2.4 Abandoned Models: DQN

Felix Exner

The initial problem can be viewed as a high dimensional regression problem for a deep neural network regarding the Q-function. Using that approach, the Model is called a Deep Q-Network (DQN).

As the game state can be represented in grid-like data, a convolutional neural network (CNN) is a natural choice. The CNN is able to process the image-like data as input and can be modelled such that it outputs scalars. This approach has been successfully implemented already in a various multiagent pursuit evasion games as which bomberman can be classified as. [Mni+13] [Sta16]

Algorithm. For developing the algorithm we followed the high level algorithm in [Mni+13]:

---

select action $\mathbf{a}$ following ϵ-greedy

$\rightarrow$ execute action $\mathbf{a}$ in emulator and observe reward $\mathbf{r}$ and subsequent state $\mathbf{s}'$

$\rightarrow$ store tuple ($\mathbf{s}$, $\mathbf{r}$, $\mathbf{a}$, $\mathbf{s}'$) in replay memory $\mathcal{D}$

$\rightarrow$ sample minibatch ($\mathbf{s}_j$, $\mathbf{r}_j$, $\mathbf{a}_j$, $\mathbf{s}'_j$) from $\mathcal{D}$

$$\rightarrow \text{ set } y_j = \begin{cases} \mathbf{r}_j + \gamma \max_{a'} \mathcal{Q}(\mathbf{s}'_j, \mathbf{a}'; \Theta) & \text{if } \mathbf{s}'_j \text{ non-terminal} \\ \mathbf{r}_j & \text{if } \mathbf{s}'_j \text{ terminal} \end{cases}$$

$\rightarrow$ perform gradient descent for Loss $\mathcal{L} = (y_j - \mathcal{Q}(\mathbf{s}_j, \mathbf{a}_j; \Theta))^2$

Here we used experience replay memory to create training data for the network $\mathcal{Q}(s)$ with parameters Θ. The network is supposed to be a function approximator for the Q-function. Unlike in the original formulation of the Q-learning process, here the Q-function only takes the state s as an input and gives as many scalars as the dimension of the action space, such that every scalar corresponds to a certain action. In this way we again have state-action pairs, as in the original formulation.

Network Architecure. Figure 1 shows a graphical representation of the network applied to the coin-heaven scenario having the objective to collect as many coins as possible. The state is represented in 3-channel grid-like data (similar to a RGB-image). As there are no crates or opponents at this stage, the three channels represent the background map (walls), the position of the agent and the position of the coins. We chose a binary representation of the grid, that is colored cells represent value 1, empty cells value 0.

The input was convoluted twice having 5 and 10 output channels after the first and second convolution, having a maxpool in between. As we need scalar outputs, the grid-like data is flattened and followed by a fully connected layer. Finally, four scalars, each representing an action-state pair, are given out. As activation function we used ReLu in every layer.

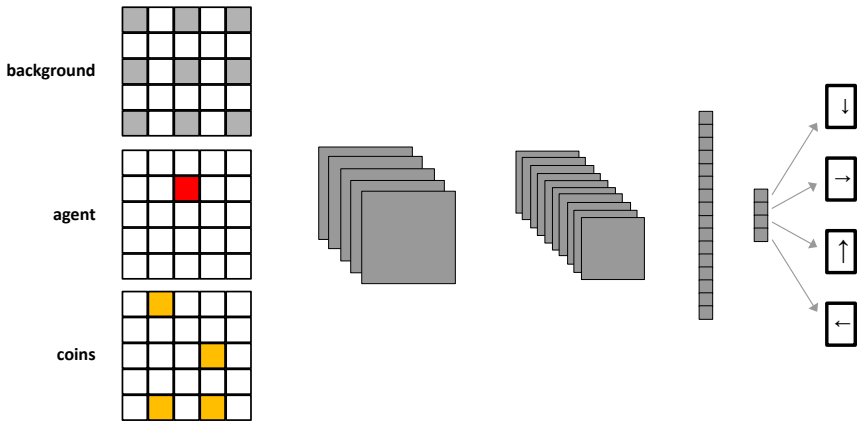


Figure 1: CNN Architecture. Two convolutional layers with maxpool followed by a fully connected layer. ReLu as activation function throughout the network.

As optimizer we used the Adam-optimizer with a learning rate of 0.01. As the criterion a mean squared error (MSE) as proposed by [Mni+13] was used. The MSE represents the time difference error and be traced back to the Bellmann equaions, predicting a time difference error of 0 when the approximation of the Q-function equals the true Q-function. Thus, minimizing the Loss equals approaching the true Q-function.

Results. Unfortunately, training the DQN for 4000 rounds showed no evidence of reasonable behaviour of the agent. What we observed was a constant maximum of the Q-function, independent of the state which led to a constant movement of the agent in one single direction. We tried different learning rates and other hyper parameters, all leading to the same behavior, making us abandon the DQN model.

In the DQN paper it is shown that they could achieve a reasonable increase of the average reward after training the network for roughly 1.500.000 frames for two Atari games. [Mni+13] As we limited one round to 100 steps during training, 4000 rounds of training correspond to 400.000 frames. It is possible that the architecture could be successful enforcing more training.

3 Design Choice

Dorina Ismaili

In order to design a robust and scalable agent capable of learning strategies in a complex environment, we specialized two main learning algorithms being Q-learning and Deep Q-Learning Network. Bomberman presents a dynamic environment, hence designing a feature engineering approach to represent the complex environment in a simplified manner is a not so trivial approach to undertake. We require the agent to make informed decision.
In our scenario, the state representation encapsulates different features like agent's position, layout of walls and crates, details about bomb information, bomb timer, proximity to enemies, proximity to bombs, power ups, coin locations, escape spaces etc. By combining these features we are able to construct a state representing the environment. We ideally reduce the complex structure of the environment, and are now able to compress the state space without losing any critical information.
Additionally, the action space consists of four crucial steps (up,right, down, left), place bombs or wait. To ensure consistency, we make sure the features are scaled and normalized. We require that actions like move toward a coin, escape from bombs, blast a bomb be observed to a maximum distance, as the far-off decisions have no impact on the decision-making. Therefore, by selecting key aspects of the feature engineering we were able to simplify the learning problem, allowing the agent to make more informed decisions and achieve reasonable performance.
Reward shaping: was used to encourage our agent to win more points when taking certain actions like when collecting coins, avoiding bombs etc. The Q-learning is implemented such that by giving extra points to the agents, we direct it toward better actions.
Limited scalabilty: As the game progresses, the state space becomes too large to be handled by the Q-learning algorithm. In our case, it is both implemented with the q-table and without a q-table. Therefore, we use another algorithm to deal with this scalability.

Additionally, to take care of the scalability limitations, we used Deep Q-Network as a second method. The DQN method uses a neural network to approximate the Q-values. This allows the agent to run among different states without implementing all of them.
Different from Q-learning, we are not required to manually engineer features, there-

fore automatically learning features and patterns.

Feature learning: the neural network in DQN learns features from raw game inputs.

Reward Shaping: Same as in Q-learning the rewards were implemented to encourage survival, penalizing unnecessary movements.

Experience Replay: The agent stores past experience(states, action, rewards) in a reply buffer to ensure efficient learning during training.

Exploration strategy: DQN uses an epsilon-greedy strategy, where epsilon controls the trade-off between exploration and expolitation. At each time step, the agent explores by selecting a random action with a probability epsilon.

Hyperparameter tuning: Given its construction, the Q-learning mixed with neural network requires to tune the parameters like learning rate, discount factor, target reward etc.

Optimization Algorithm: are commonly used on the gradient descent algorithm in order to accelerate convergence.

Higher performance: The agent develops more sophisticated strategies for survival,like placing bombs, and collecting rewards, therefore leading us to a higher performance and convergence.

Nevertheless, the implementation has shown not very reasonable results for the neural network, while the Q-function results have been more accurate. Given our implementation, the agent's performance was not distinguishable, hence we explored more on the Q-learning algorithm.

4 Agent Training and Development

Suryansh Chaturvedi

4.1 Training Process

Suryansh Chaturvedi

The core of our project lies in training our Q-learning agent to effectively play Bomberman. This involved an iterative process of refining both the agent's features and the associated reward structure to guide its learning. We utilized the provided Bomberman game environment, which allowed us to simulate games, observe the agent's actions, and provide feedback in the form of rewards.

4.1.1 Iterative Refinement

Our training methodology revolved around an iterative process:
Feature Engineering: We started with a basic set of features (as described in Section 2.4) that allowed the agent to interact with the environment at a fundamental level.
Training: We trained the agent for a set number of rounds, observing its performance through the chosen evaluation metrics.
Analysis: We analyzed the agent's behavior, identifying weaknesses and areas for improvement.
Feature and Reward Refinement: Based on our analysis, we either added new features (like the "Distance to Safe Tile" feature) to address specific shortcomings or adjusted the rewards associated with certain events to encourage or discourage particular actions.
Repeat: We repeated steps 2-4, iteratively refining the features and rewards until we achieved a satisfactory level of performance.

4.1.2 Training Phases

We divided the training process into three distinct phases to systematically build the agent's capabilities:
Coin Heaven: This initial phase focused on the core mechanics of navigation and coin collection. The agent's action space was limited to movement only, allowing us to establish a strong foundation for basic navigation.
Crates Introduction: In the second phase, we introduced crates into the environment and expanded the action space to include bomb placement and waiting. The

focus here was on developing the agent's ability to strategically destroy crates to collect coins while avoiding self-destruction from bomb blasts.

Opponent Integration: The final training phase introduced other agents into the game, adding a significant layer of complexity. The agent had to learn to navigate and place bombs effectively while considering the actions and positions of its opponents.

4.2 Training Data and Feature Evolution

Suryansh Chaturvedi

To illustrate the impact of feature engineering and hyperparameter tuning, we present a snapshot of our training data, focusing on how the agent's behavior evolved in response to changes in its feature set and reward structure.

4.2.1 Coin Heaven: Early Stages

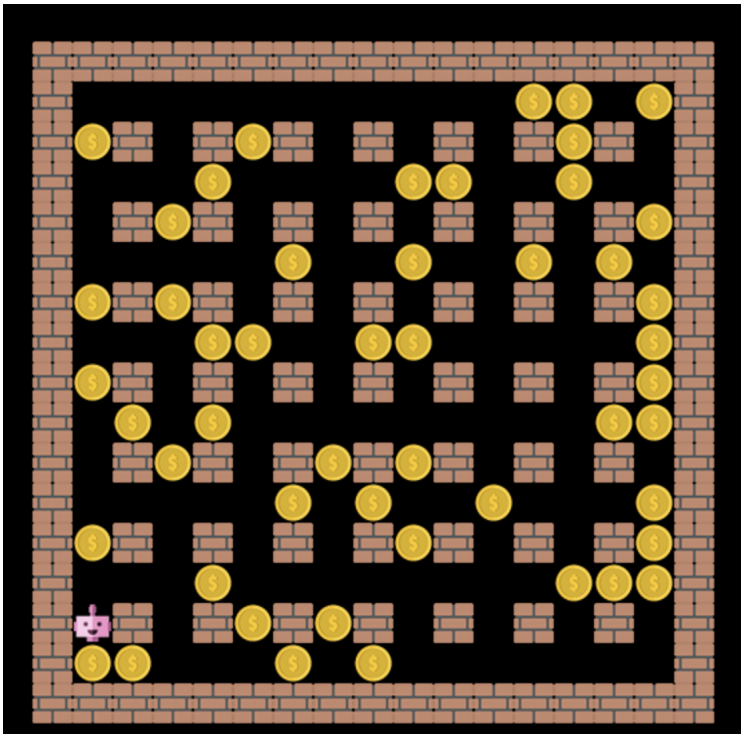


Figure 2: Coin Heaven Scenario

In the early stages of training in the "Coin Heaven" scenario (Figure 2), the agent's behavior was largely random. The table below shows the activation frequency of relevant features and the average reward obtained during this initial phase.

Feature	Activation Frequency	Average Reward with Activation
Moves Towards Nearest Coin	55%	+8
Valid Move	92%	+2
Collects Nearest Coin	8%	+50
Drops Bomb Near Crate	12%	+60
Moves Towards Nearest Crate	30%	+3
Moves to Avoid Bomb	25%	+15
Exits Bomb Range	18%	+1500
Safe Bomb Placement	10%	+80

Table 1: Feature Activation and Reward Data

As you can see, the agent was moving validly most of the time, but its success in approaching and collecting coins was limited. This was due to the initial random nature of its actions.

4.2.2 Coin Heaven: Improved Performance

After several training rounds and the addition of a feature specifically rewarding coin collection, the agent's behavior drastically improved. The updated table below reflects this improvement:

Feature	Activation Frequency	Average Reward
Distance to Nearest Coin	45%	+5
Valid Move	95%	+1
Collecting Coins	10%	+50

Table 2: Feature Activation and Reward Data

The agent's ability to approach and collect coins significantly improved, resulting in a substantially higher average reward.

4.2.3 Crates Introduction: Challenges and Refinements

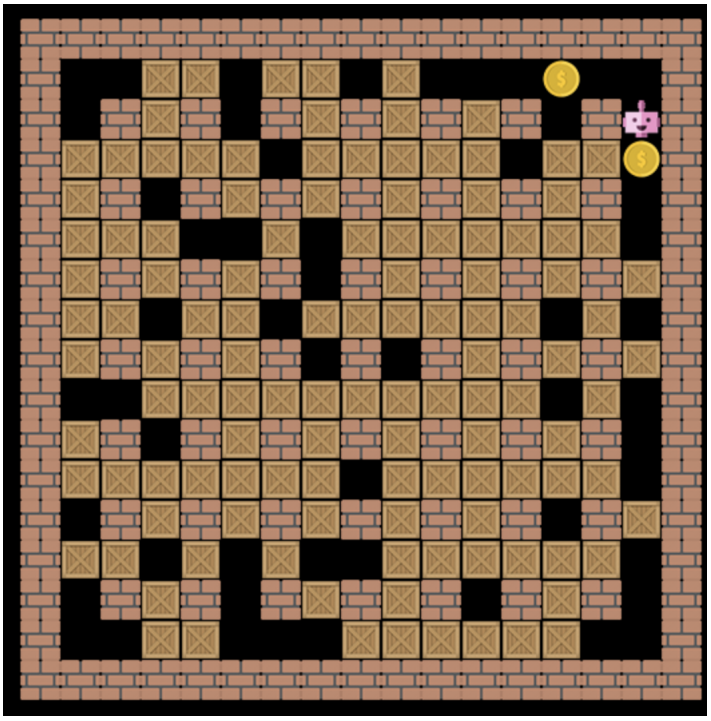


Figure 3: Classic Bomberman Scenario with Crates

With the introduction of crates (Figure 9) and the expanded action space, the training process became more challenging. Initially, the agent struggled to balance bomb placement with self-preservation. The table below shows the initial performance after introducing crates:

Feature	Activation Frequency	Average Reward
Distance to Nearest Coin	35%	+4
Valid Move	92%	+1
Collecting Coins	8%	+40
Approaching Crates	50%	+2
Placing Bomb Near Crate	35%	+10
Moving From Bomb	15%	+5
Is in Bomb Blast Radius	25%	-500

Table 3: Initial Feature Activation and Reward Data with Crates

The agent placed bombs frequently but often found itself caught in the blast, leading to a lower average reward.

4.2.4 Crates Introduction: Improved Bomb Avoidance

To address this issue, we introduced the "Distance to Safe Tile" feature and significantly increased the penalty for "EXPLOSION_HIT". This encouraged the agent to prioritize finding a safe spot before placing a bomb. The effect is evident in the updated table below:

Feature	Activation Frequency	Average Reward
Distance to Nearest Coin	45%	+5
Valid Move	96%	+1
Collecting Coins	15%	+80
Approaching Crates	30%	+3
Placing Bomb Near Crate	25%	+70
Moving From Bomb	40%	+30
Distance to Safe Tile	70%	+10
Is in Bomb Blast Radius	5%	-1500

Table 4: Improved Feature Activation and Reward Data with Crates and Safe Tile Feature

The agent's bomb avoidance significantly improved, leading to a much higher survival rate and, consequently, a higher average reward.

4.2.5 Opponent Integration: Adapting to Adversaries

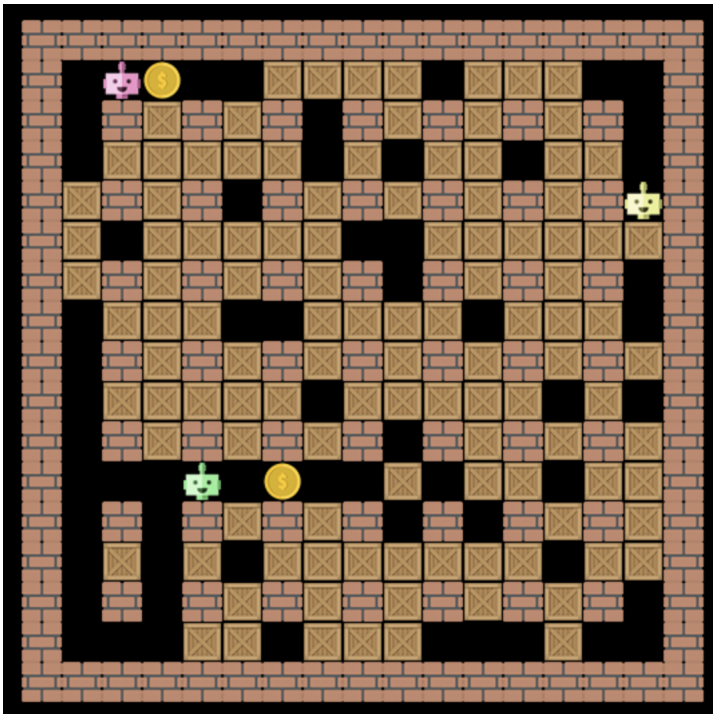


Figure 4: Multi-Agent Bomberman with Opponents

Integrating opponents into the training process (Figure 4) presented the agent with the most challenging scenario. It had to learn not only to navigate, collect coins, and place bombs effectively but also to anticipate and react to the actions of other agents. We observed a significant drop in performance initially.

Feature	Activation Frequency	Average Reward
Distance to Nearest Coin	25%	+3
Valid Move	90%	+1
Collecting Coins	5%	+60
Approaching Crates	40%	+2
Placing Bomb Near Crate	20%	+30
Moving From Bomb	30%	+15
Distance to Safe Tile	50%	+8
Is in Bomb Blast Radius	15%	-1000
Distance to Nearest Enemy	60%	-5
Can Place Bomb Near Enemy	10%	+40

Table 5: Initial Feature Activation and Reward Data with Opponents

The agent struggled to survive and effectively place bombs near opponents.

4.2.6 Opponent Integration: Learning to Compete

By adjusting rewards to encourage offensive bomb placement and by refining the "Distance to Nearest Enemy" feature, we saw significant improvement.

Feature	Activation Frequency	Average Reward
Distance to Nearest Coin	35%	+4
Valid Move	94%	+1
Collecting Coins	10%	+70
Approaching Crates	35%	+2
Placing Bomb Near Crate	20%	+65
Moving From Bomb	40%	+25
Distance to Safe Tile	60%	+9
Is in Bomb Blast Radius	8%	-1200
Distance to Nearest Enemy	70%	+10
Can Place Bomb Near Enemy	45%	+120

Table 6: Improved Feature Activation and Reward Data with Offensive Bomb Placement

Our training results highlight the effectiveness of iteratively refining features and rewards in reinforcement learning to achieve increasingly better performance in complex game scenarios.

5 Experiments and Results

5.1 Feature Engineering

Felix Exner

This section outlines the features we used to represent the game state and the process developing them. We will provide justification for the selected features and classify them.

Features represent the relevant characteristics of the environment's state that the agent uses to make decisions, simplifying and capture the most important aspects of the environment. Choosing a proper state representation is critical to the learning process of the agent. Thus, the process if feature engineering is crucial in developing a Q-learning model. As Q-learning uses a state-action representation for the Q-function, we have a unique feature representation of the state for every action, see figure 5. The algorithm now is designed such that these feature vectors are multiplied element-wise by weights which have to be learned. Finally, the sum is taken to obtain the Q-values.

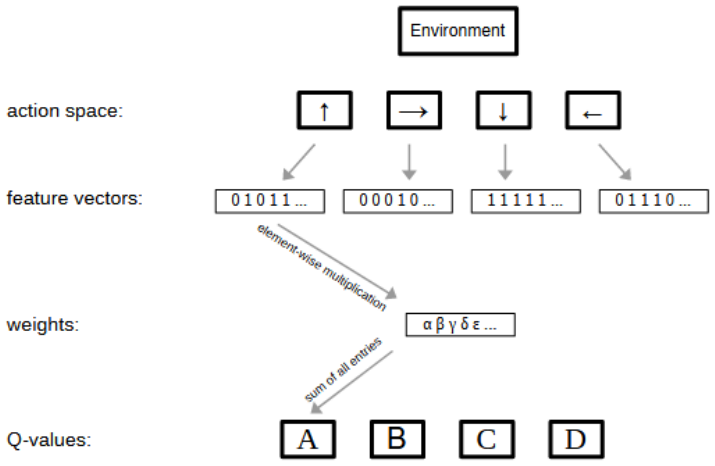


Figure 5: Flowchart of the algorithm. Features for each action-state pair get multiplied by model weights resulting in scalat Q-values.

General Approach: As our approach to feature engineering we chose an iterative process. We began by focusing on the "Coin-Heaven" scenario with a limited action space—restricted to moving up, right, down, and left. The primary objective was collecting coins. During this phase, we implemented the core features and initiated the model training process. After reviewing the results, we identified additional features that could improve performance and implemented them into the model,

followed by another round of training.
Once we achieved satisfactory results in the "Coin-Heaven" scenario, we advanced to the "Classic" scenario, introducing the full action space by adding the options to wait or drop a bomb. Initially, we trained the model without opponents, focusing only on handling a map with crates. After achieving a stable performance, we introduced opponents into the environment, completing the feature set for this phase of development.

5.1.1 First Stage: Coin-Heaven

Approaching Coins: Starting with the "Coin-Heaven" scenario the most important issue for an efficient behavior of the agent was to make it to move towards the nearest coin while avoiding walls. We created a function which takes the `game_state` variable and an action as input and gives a binary output.
The function reads out the agents and nearest coins position from the `game_state` and calculates the subsequent position of the agent following the action. Using ordinary euclidean metric, it determines if the new position is closer to the nearest coin than the previous one. We also added a feature that ensures that there is neither a wall or a crate at the new position based on checking the value of `game_state['field']`. Depending on these conditions, a binary output is given. Thus, a feature representing whether the action-state pair causes the agent to get closer (=1) to the nearest coin or not (=0) is created.
Collecting Coins: To make sure that the agent strives for not only approaching but collecting the nearest coin, a feature was added which represents if a given action leads to that event. Again, the agents and the nearest coins position is read out and the subsequent position of the agent is determined. Now, based on the condition if the subsequent position corresponds to the position of the nearest coin, a binary output is given. Thus, a feature representing whether the action-state pair causes the agent to collect the closest coin (=1) or not (=0) is created.

5.1.2 Second Stage: Crates

Using the reduced action space and the Coin-Heaven scenario, a reasonable behaviour of the agent has been achieved leading us to continue including crates into the game. For now, we still did not include opponent agents. The situation changed as now we used the full action space including dropping bombs and waiting. At this stage the objective of the agent was destroying crates and collecting contained coins without bombing himself. Thus, we implemented features leading the agent to be

aware of crates and bombs.

Approaching Crates: We decided our agent only to be aware of the nearest crate, creating a function which detects the crate which has the smallest euclidean distance to the agent. Very similar to the function which detects whether a move brings the agent closer to a coin, we implemented a function which creates a feature representing if an action brings the agent closer to the nearest crate (=1) or not (=0).

Destroying Crates: To destroy a crate, the agent needs to place a bomb near it. We created a function which checks if an action will lead to placing a bomb at max three positions next to a crate. As bomb explosions only propagate through free space, we had to ensure that there is no wall in between the agents position and the target crate. Therefor, we utilized that the x and y coordinates of walls are always odd. Thus, if the agent is located at an even x coordinate, there are no crates in y-direction and vice versa. Thus, placing a bomb at an even x-direction, the explosion can propagate in y-direction freely.

Avoiding Bombs: As bomb explosions can not only destroy crates but also kill the agent itself, it was necessary to enable the agent to learn to avoid bombs. Here, we chose a balancing reward system meaning we implemented a feature representing the agent to approach a bomb to give a positive reward and a feature which detects the agent moving away from a bomb to give negative reward.

To consider bombs which could kill the agent only, we constructed a function which detects if a bomb is possibly dangerous for the agent. This is either the case if the agent stands exactly upon the bomb, or is located at max 3 fields off in x or y-direction from the bomb and if there are no walls in between the agent and the bomb. Therefor we again utilized the fact that walls are located at odd coordinate indices only.

Originally we planned to consider more than one dangerous bombs, but dropped that later as the code gets much more complex. This is why also created a function which sorts up to three dangerous bombs by euclidean distance with respect to the agent. We continued by considering the nearest bomb only. The function works very similar to the one which creates the feature resembling moving towards a coin. It takes the respective action, calculates the subsequent position of the agent and checks if the distance to the bomb got larger (=1) or not (=0). The function creating the feature resembling approaching a bomb works the other way around.

Crate Traps: Having these essential features implemented, we started to train the agent in the classic scenario. After reasonable training, we observed the behavior of the agent. One heavy problem we detected was that after placing a bomb, the agent

moved in a direction which was deadlocked by crates such that the agent could not move as far away from the bomb as it would have been necessary to escape the explosion range and died. We call this a crate trap, see fig. 6.

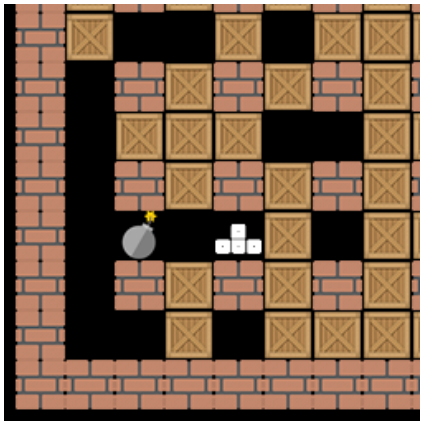


Figure 6: The agent (represented by arrow keys) moved into a crate trap trying to escape the bomb. This leads to certain death.

Thus, we implemented a function to create a feature resembling if an action leads the agent moving into a crate trap to assign a negative reward on this event. The function takes the agents and the nearest bombs position and checks if the movement leads to moving away from the bomb, since entering crate traps has been observed while the agent tries to avoid a bomb.
To check if the subsequent position following the action is a crate trap, the function checks if the field value next to the subsequent position is free (=0). If this is not the case, the function returns the value 1, resembling a crate trap, otherwise 0.
Safe Start: Another problem we faced was that the agent frequently chose to place a bomb as the first action in step 1. In most of the cases that led to certain death because the agent was crate trapped in every possible direction of movement, see fig. 7.
Consequently, we created a feature which represents if the respective action in step 1 leads to certain suicide. The function creating this feature first reads out in which of the four possible corners the agent spawns. If the agent spawns e.g. in the lower right corner, safe and valid actions are moving 'UP' and 'LEFT'. Thus, in this case, the function creates a feature 1 for both of these actions and 0 for all the others.
We imposed a high penalty if the agent chose a not-safe action in the first step. Training the model resulted in the agent to avoid these actions in the beginning of the game.
Bombs still problematic: As the agent now learned to survive the start of the

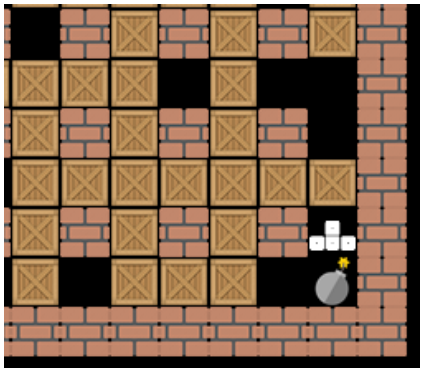


Figure 7: Starting in the lower right corner, the agent (represented by arrow keys) placed a bomb as the first action. This leads to certain death in most of the cases as it is crate trapped.

game, we did explore the behavior further. What we still observed was an unsatisfactory behavior of the agent in regard to dying by bomb explosions. That led us to implement two more features hoping to avoid this.

First, We implemented a function analyzing if a move leads the agent to leave the range of a future bomb explosion if there is a dangerous bomb around. Therefor the agents position is read out and the subsequent position following the respective move is calculated. If the subsequent position is out of the bombs explosion range, the function gives output 1, if not output 0. Also it makes sure that the subsequent position is a free coordinate checking its value in the same way as the former functions.

Second, we implemented a function that analyzes if a move leads to enter a current bomb explosion. Therefor we utilized the given explosion map of the **game_state** variable. Again, the function calculates the subsequent coordinates with respect to the action and checks the value of the explosion map of this coordinates. If the value represents a current explosion, it outputs 1, if not 0.

Having these features implemented, we could assign high negative reward if the agent moves into an explosion range and high positive reward if the agent exits an explosion range. Training showed an improved behavior of the agent not dying that frequently as before.

Unnecessary Bombs: Exploring the behavior of the agents further, we observed that it often placing bombs at coordinates not leading to any positive outcome. Thus, the agent unnecessarily imposes itself to a dangerous situation. To prevent this, we implemented a feature representing if placing a bomb has any positive outcome or not. This is realized utilizing the function which determines if the agent placed a bomb near a crate and just reversed its output. Thus, we could densify

the rewards assigning negative reward for placing an unnecessary bomb. Later, we expanded this function to include also near agents, not crates only.

5.1.3 Third Stage: Opponents

Suryansh Chaturvedi

With crates integrated into the model, the final stage of feature engineering involves incorporating opponent agents into the environment. The agent now has to adapt its strategy, considering the actions and positions of its adversaries, making the situation much more complex. We introduced three new features to improve the performance of the agent:

Can Place Bomb Near Enemy: This binary feature activates if placing a bomb at the agent's current position would place an enemy agent within the bomb's blast radius. It encourages the agent to use bombs offensively against opponents.

Distance to Nearest Enemy: This feature calculates the Euclidean distance between the agent and the nearest enemy agent, allowing the agent to be aware of nearby threats and act accordingly.

Valid Move (Update): We added an additional condition to the function which creates the feature "Valid move" to make sure that the subsequent position of the agent does not correspond to the position of an enemy agent.

With the complete set of features implemented, the agent can now learn to play the game in a way that considers all the essential elements: coins, crates, bombs, and opponents.

5.2 Performance in Test Scenarios

Suryansh Chaturvedi, Felix Exner

To comprehensively evaluate the capabilities of our trained Q-learning agent, we tested it in various game scenarios, each designed to assess specific aspects of its gameplay.

5.2.1 Coin Heaven

The "Coin Heaven" scenario serves as a benchmark for the agent's fundamental ability to navigate the environment and collect coins effectively. The agent's performance in this scenario provides insights into its efficiency in finding and collecting coins while avoiding obstacles.

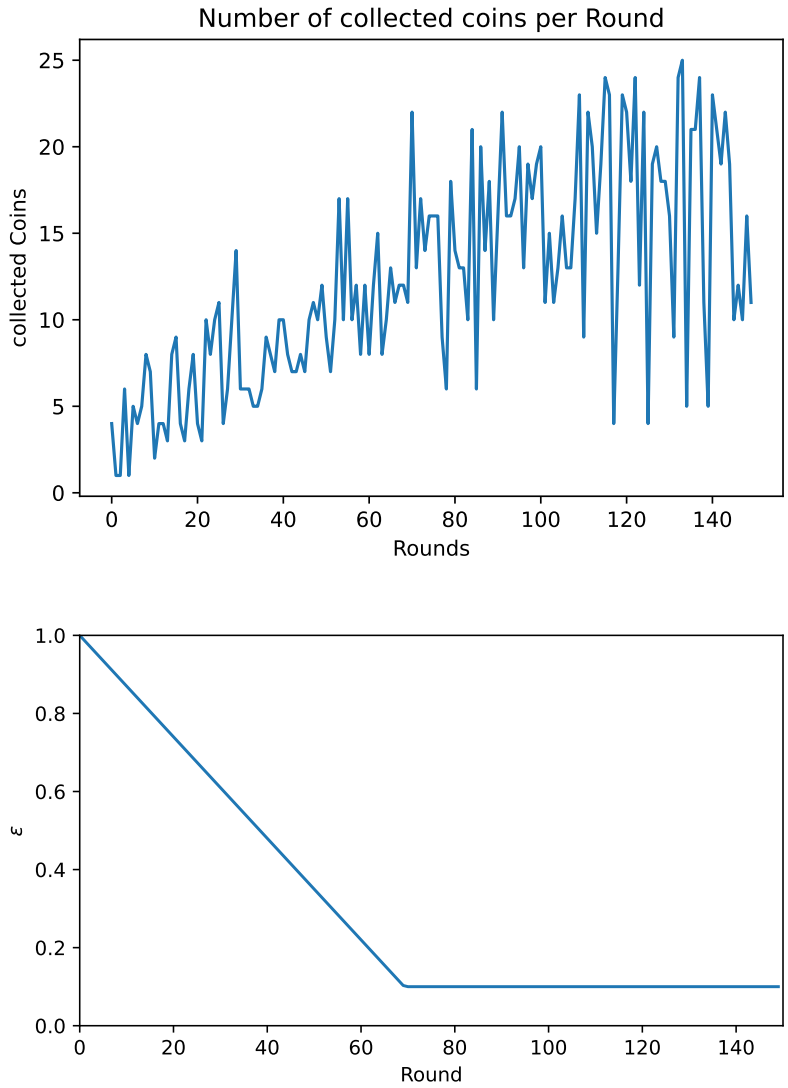


Figure 8: Development of the agents ability of collecting coins over 150 rounds of training.

Results: The agent achieved an average score of 42 points per game in the "Coin Heaven" scenario, with a coin collection rate of 82%. It consistently survived for the entire duration of the game (100 steps). Figure 8 (top) shows a training cycle for 150 rounds. A linear ϵ-decay according to Figure 9 (bottom) has been applied. One can observe a linear improvement of the ability to collect coins roughly following the inverse of the ϵ-curve.

Analysis: The agent showed strong performance in navigating the simple map and collecting most available coins. This indicates it has effectively learned basic movement and coin seeking behavior.

5.2.2 Crates and Hidden Coins

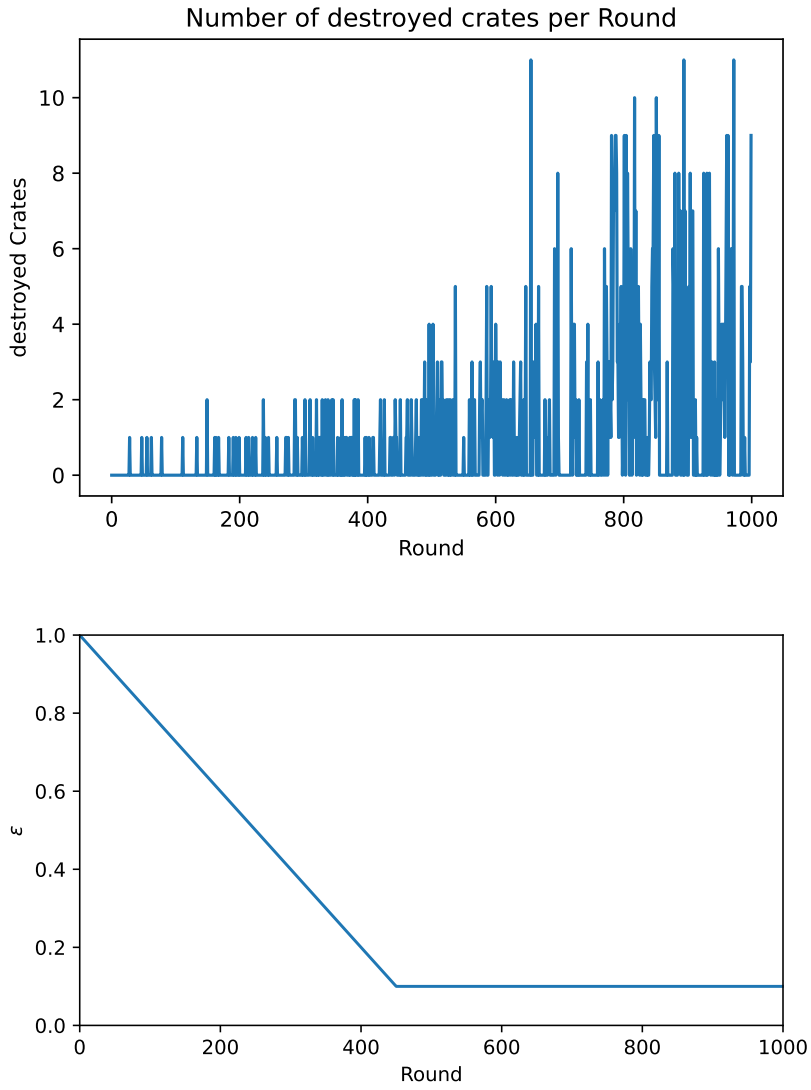


Figure 9: Development of the agents ability of destroying crates over 1000 rounds of training.

This scenario introduces crates into the environment, requiring the agent to employ bomb placement strategies to uncover hidden coins. The agent's performance here assesses its ability to balance crate destruction with coin collection while avoiding bomb blasts.

Results: The agent's performance was mixed in this scenario. It achieved an average score of 33 points per game, destroying an average of 4.5 crates per game. However, its bomb placement efficiency was only 30%, indicating room for improvement in strategic bomb placement.

Figure 9 (top) shows the agents development regarding the ability of destroying crates over 1000 rounds of training. We used an ϵ-decay according to Figure 9 (bottom) and limited the ϵ to 0.1. Here, we modified the steps per round to 100. One can clearly see a positive development starting at an average number of only 0.5 destroyed crates for the first 300 rounds up to the final 4.5 destroyed crates on average.

Analysis: While the agent understands the concept of placing bombs near crates, it seems to struggle with optimizing placement for maximum crate destruction. Further feature engineering to better assess crate density and potential blast impact could enhance the agent's effectiveness.

5.2.3 Opponent Interaction

This scenario introduces opponent agents, challenging the agent to compete in a dynamic and adversarial environment. The agent's performance in this scenario evaluates its ability to survive, collect coins, and utilize bombs offensively against other agents.

Results: In the multi-agent scenario, the agent struggled significantly. It often got trapped in dangerous situations, leading to a low average survival time of 40 steps and a low average score of 12 points. Its win rate was also low at 10%.

Analysis: The significant performance drop in the multi-agent environment indicates that the agent lacks the necessary skills to effectively anticipate and react to opponent actions. It often placed bombs in disadvantageous positions, leading to self-destruction or being caught in the opponent's blasts.

5.2.4 Rule-Based Agent Challenge

To further assess our agent's competence, we challenged it against a rule-based agent provided in the project framework. This benchmark allows us to compare our Q-learning agent's performance against a baseline strategy based on predefined rules and heuristics.

Results: Our agent won only 20% of the games against the rule-based agent, achieving an average score of 40 points.

Analysis: The poor performance against the rule-based agent suggests the agent's learned strategy is not yet refined enough to compete effectively. The rule-based agent's superior performance likely stems from its ability to anticipate bomb explosions and navigate safely, a skill our Q-learning agent needs to further develop.

6 Challenges and Limitations

Suryansh Chaturvedi

While our Q-learning agent demonstrated promising results, we encountered several challenges and limitations during the development and training process:

State Space Complexity: The Bomberman environment presents a large and complex state space, making it computationally expensive to store and update a comprehensive Q-table. Feature engineering helped reduce this complexity, but the curse of dimensionality remains a concern, especially as the agent's strategies become more sophisticated.

Reward Shaping: Designing an effective reward structure proved challenging, particularly when balancing offensive and defensive actions. Fine-tuning the reward values for events like bomb placement, coin collection, and opponent interaction was crucial to achieving the desired behavior.

Generalization to Unseen Scenarios: While the agent performed well in the trained scenarios, its ability to generalize to novel game maps or against unknown opponent strategies remains limited. This highlights the importance of further research in domain adaptation and transfer learning for reinforcement learning agents in dynamic game environments.

Limited Exploration: Despite employing an epsilon-greedy strategy, the agent might not have fully explored the vast state-action space, potentially missing out on optimal strategies. More sophisticated exploration techniques, such as optimistic initialization or curiosity-driven exploration, could be investigated in future work.

7 Future Work

Suryansh Chaturvedi

Building upon the foundation laid in this project, we propose several directions for future work:

Advanced Feature Engineering: Incorporate more sophisticated features to capture complex game dynamics, such as bomb blast predictions, opponent behavior modeling, and strategic power-up usage. This could involve using relative distances, temporal information, or game-specific heuristics.

Deep Reinforcement Learning: Explore the use of deep neural networks to approximate the Q-function, potentially improving the agent's ability to learn from high-dimensional state representations. This would involve implementing a Convolutional Neural Network (CNN) based DQN to process the grid-like game state

information.

Multi-Agent Coordination: Investigate techniques for multi-agent coordination, allowing our agent to collaborate with other agents in team-based Bomberman scenarios. This could involve communication protocols, shared reward structures, or decentralized learning algorithms.

Opponent Modeling: Develop techniques to model opponent behavior, enabling the agent to anticipate and counter their actions more effectively. This could involve techniques from game theory, opponent modeling algorithms, or adversarial training methods.

Transfer Learning: Investigate transfer learning methods to enable the agent to adapt to new maps or opponent strategies more quickly by leveraging knowledge learned from previous experiences. This could involve pre-training on a diverse set of maps or using techniques like fine-tuning or domain adaptation.

References

[WD92] Christopher Watkins and Peter Dayan. "Technical Note: Q-Learning". In: *Machine Learning* 8 (1992), pp. 279–292.

[Mni+13] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG]. URL: https://arxiv.org/abs/1312.5602.

[Sta16] Maxim Egorov Stanford. "Multi-Agent Deep Reinforcement Learning". In: 2016. URL: https://api.semanticscholar.org/CorpusID:265700928.

[KDW18] J.G. Kormelink, M.M. Drugan, and M.A. Wiering. "Exploration methods for connectionist Q-learning in bomberman". In: *Proceedings of the 10th International Conference on Agents and Artificial Intelligence* (2018).

[SB18] Richard S. Sutton and Andrew G. Barto. "Reinforcement Learning: An Introduction". In: *The MIT Express* (2018).

[Aut21] E. Author5. "Evolutionary Algorithms for Bomberman". In: *Yet Another Journal* 3.3 (2021), pp. 21–30.

[Tri21] Ngo Hung Minh Triet. "Intelligent Bomberman with Reinforcement Learning". In: *35th Twente Student Conference on IT* (2021).

[AA22] C. Author3 and D. Author4. "Monte Carlo Tree Search for Bomberman". In: *Another Journal* 2.2 (2022), pp. 11–20.

[Kow+22] Dominik Kowalczyk et al. "Developing a Successful Bomberman Agent". In: (2022).