Reinforcement Learning for Bomberman
Final Project

Angelina Basova
Sven Zelch

April 4, 2022

Tutor:	Lars Erik Kuehmichel
Instructor:	Ullrich Köthe
URL:	https://github.com/SeZven/Terminator

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

In the following section we state the problem and give an overview of the theory we used for solving it.

1.1 Problem definition Sven Zelch

In this report we are developing an agent for the game Bomberman. In Bomberman four agents compete against each other with the goal to reach the highest score of those four agents and by that winning the game. The game is played in a 17x17 match field where each agents starts in a random corner. The corner an agent starts can not be the same as the corner of another agent.

An agent can perform one action in each discrete time step. In other words every agent plays at the same speed and can not perform two actions while other agents only perform one. It can choose from six actions. An agent can either move horizontally (left or right), vertically (up or down), drop a bomb or wait.

The match field consists of empty fields, walls, coins, crates and agents. One agent can only walk through empty fields and coins, but can not walk through walls, creates and other agents. When an agent reaches a coin field the coin gets collected and the agent's score increases by one point. Afterwards, the coin field turns into an empty field and other agents can not obtain a coin on this field anymore. Another way for the agent to increase its score is by blowing up other agents, which is rewarded with 5 points. Some coins are visible on the field while others are randomly hidden in crates and have to be found by blowing up crates. These coins can be collected as described above and also get a score increase by 1.

Something really important about the game is that each agents only have 0.5 seconds to chose an action. In case an agent doesn't choose an action within the 0.5 seconds, then the action wait is executed. Additionally, the agent will get punished for taking too long to choose an action in the following time step. The exceeding think time of the agent will be subtracted from the next time step.

1.2 Reinforcement Learning Sven Zelch

Reinforcement Learning is typically used for gaming and that is why it is predestined for our problem. The agent collects data, in our case q-values which we will talk about later, and makes predictions about how good a certain action is given the current state. A speciality of Reinforcement learning is that the agent can change the environment, the match field in our case, by certain actions, e.g. placing a bomb and blowing up a crate.

Also feedback in Reinforcement Learning is only given at the end of a game. Which seems problematic in a game like Bomberman where it takes a lot of steps until a game is finished and the agent just gets the reward according to its placement at the end of the game. We struggled with this a lot, but especially reward shaping help us with speeding up the learning process. Reward shaping means designing rewards for moves that we think are good like moving away from a bomb or moves that are bad like staying in bomb range. We described it as moves we think are good, because they do not need to end up being good. For example placing a bomb near an enemy agent is good, but not if gets into an position where the agent can not run away from the bomb and blows himself up. A more detailed explanation of the reward shaping we did by adding auxiliary rewards can be seen at section 3.1.1.

1.3 Markov Property and Markov Decision Process Angelina Basova

The Markov property for a reinforcement learning problem states that the environment's response at time t+1 depends only on the state and action representations at time t [4]. Formally:

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

where s' is the state at time t+1, r is the reward at time t+1, s is the state at time t and a is the selected action at time t. The Markov property applies to all rewards r, states s', states at time t S_t, and action space A_t. This property is essential for this report because decisions and values are assumed to be a function only of the current state.

A Markov decision process is a decision-making model [3] that sat-

isfies the Markov property [4]. A Markov decision process is a tuple:

$$M = (S, A, r, p)$$

where S is the state space, A is the action space, $r : S \times A \longrightarrow \mathbb{R}$ is the expected reward function and $p : (s'|s, a) \longrightarrow [0, 1]$ is the transition probability that action a in state s will lead to state s' [4], [3]. The goal of a Markov decision process is to find the optimal policy, which maximizes the accumulated reward [4]. When the transition probabilities are not known, the optimal policy can be learned with Q-Learning.

1.4 Q-Learning Angelina Basova

Q-Learning is a model-free, reinforcement learning algorithm [4]. Model-free means that the agent learns a good or optimal policy without learning the transition probabilities associated with the Markov decision process [4]. Hence, it doesn't require a model of the environment. Q is a learned action-value function, that approximates the optimal policy. The following equation defines the update after the transition to a new state:

$$Q(s, a) = Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

where α is the learning rate and γ the discount factor. In order to converge to the optimal Q^* three conditions need to be fulfilled [5]. First, each state-action pair should be experienced an infinite number of times. Second, the rewards should be bounded. Third, the exploration and learning rate reduce to zero.

The Q-Learning algorithm is shown in procedural form in Algorithm 1. To find the optimal policy, exploration and exploitation is used with the ϵ-greedy policy. The ϵ-greedy policy is highlighted in Algorithm 1 with a light blue color. The hyperparameter ϵ is a uniform random number between zero and one, where a value of one leads to exploration of random actions and a value zero leads to exploitation of previous knowledge.

Algorithm 1 Q-Learning algorithm with ϵ-greedy policy

Initialize $Q(s,a), \forall s \in S, a \in A(s)$, arbitrarily, and Q(terminal-state, $\cdot$) = 0
repeat(for each episode):
  repeat(for each step of episode):

    $n \leftarrow$ uniform random number between 0 and 1
    if $n < \epsilon$ **then**
      $A \leftarrow$ random action from the action space
    else
      $A \leftarrow \arg\max_a Q(s,a)$
    end if
    Take action A, observe R, S'
    $Q(s,A) \leftarrow Q(s,A) + \alpha[r + \gamma \max_a Q(s',a) - Q(s,A)]$
    $S \leftarrow S'$
  until S is terminal
until

1.4.1 Reward Function

Potential-based reward shaping has shown to significantly improve agent performance [2]. The idea is to modify rewards by a potential function to account for problems such as agents moving in cycle [2]. An extension of the static reward shaping is the dynamic potential-based reward shaping, which also takes into account the time step of the reward in the potential function Φ [1]. It is formally defined as:

$$F(s,t,s',t') = \gamma\Phi(s',t') - \Phi(s,t)$$

where the potential function Φ is some function over states and time, s is the state at time t and s' is state at time t'. Note that time t' is the time step after time step t or in other words $t' = t + 1$.

The total reward R' for an action is the sum of the rewards and the dynamic potential function, or in other words:

$$R' = R + F(s,t,s',t')$$

.

2 Methods

Subsequently we talk about the two methods we used for training our agents. First, we introduce our implementation of the Q-Learning algorithm. Then we introduce the deep Q-Learning implementation.

2.1 Q-Learning

The following subsection contains the implementation details of the Q-Learning algorithm. First, we define the Q table size problem, then we introduce the selected data structure for the Q-Table and outline the advantages. Next, we illustrate feature structure of a state. Finally, we describe the need to compute similar states and define the logic behind the similarity function.

2.1.1 Q-Table size problem Sven Zelch

At first we set up a q-table with every possible state action pair and came to the conclusion that it is way to big and we will have issues later, e.g. training would take way to long and it is unlikely that the agent will discover every state action pair during training. As a solution to this problem we taught of shrinking the environment in order to get a smaller q-table, which makes not discovering every state action pair less likely, however the q-table was still to big and although it was less likely to have not discovered state actions pairs during training it is still very likely that many of those occur. Since we came to the conclusion that we would have way to many entries inside the q-table if we would consider all state action pairs even in the smaller environment, we considered using a space-matrix approach. For this approach we did us a dictionary for easier readability, especially for later debugging and reward shaping with auxiliary rewards in mind.

2.1.2 Q-Table Angelina Basova

We initialize the Q table as a dictionary, whose keys are the features and the values are the values of the Q function for the six actions. Choosing a dictionary as the preferred data structure for the Q table provided us two major benefits. First, we we were able to increase its size and thus its feature space flexibly. This proved to be quite helpful

to derive training strategies because we could focus on the q values of almost the same states. Additionally, we didn't feel the need to create a Q table covering every possible feature space as for unseen states, we picked actions from a similar seen state.

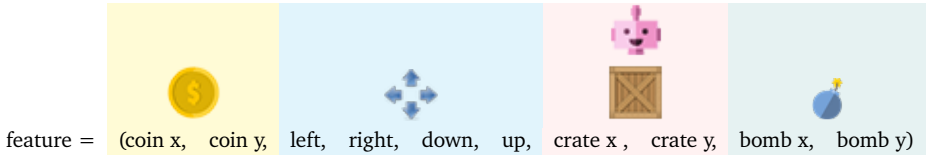
2.1.3 Feature Structure Angelina Basova

The agent receives the state as a feature consisting of ten values. Figure 1a shows the structure of a feature. The feature contains data about the nearest coin, the environment around the agent, the nearest crate or opponent and finally the nearest bomb. Specifically, the data to the nearest coin, crate or opponent and the bomb is the Manhattan distance between the agent and the object.

At the 6th and 7th element, we chose to show the distance to either the crate or the opponent because the desired behavior is the same. For both objects the agent should move next to the object to place a bomb and then move away from the bomb. As long as crates exist, the distance to the nearest crate is computed, otherwise the distance to the nearest opponent.

The environment shows which cells around the agent are free. A free cell contains the value one, while an occupied cell contains the value zero.

Figure 1b illustrates a possible feature. The feature shows that the agent is only one cell away from a coin in the x-axis as the first value equals to one. The y value equals to zero, which means that the agent is in the same row as the coin. The cells below and above the agent are free as only the down and up elements contain the value one. Furthermore, there is a crate or an opponent five cells in the x-axis and one cell in the y-axis away.



(a) Illustration of the feature structure encoding the state

feature = (1, 0, 0, 0, 1, 1, 5, 1, 9, 12)

(b) Example of a feature

Figure 1: State as a feature

2.1.4 Computation of similar states Angelina Basova

Our state representation is big considering that it contains 10 elements. As we didn't expect our agent to visit every possible state and our Q table only contains visited states, we expected to see states in the tournament that are unknown to the agent. To still allow the agent to perform well in unseen states, the agent chooses an action based on another similar state. We define similarity as the a weighted euclidean distance, with the following weights w:

$$w = (3, 3, 2, 2, 2, 2, 1, 1, 4, 4)$$

The most important criterion for the decision-making is the ability to collect points. Therefore, the feature elements showing the distance to a coin and a bomb have the highest weight. The bomb distance contains a weight of four and the coin distance contains a weight of three. The next most important elements are those of the environment, which contain a weight of one. Finally, the least important elements are the ones to the distance to crate or and opponent with a weight of one.

3 Training

This section contains training-related details about the two implemented agents. The first subsection focuses on the Q-Learning im-

plementation. The second subsection focuses on the deep Q-Learning implementation.

3.1 Q-Learning

In the following section we present the auxilliary rewards in general and the logic behind their definition. Then we define selected rewards. Finally, we show the reward shaping function and the total reward for the agent.

3.1.1 Auxilliary Rewards Angelina Basova

In order to incentivize the agent to perform beneficial actions we used a number of auxilliary rewards. The rewards were split into five different categories, which indicate in which case the auxilliary reward is earned. For instance, the auxilliary reward MOVED_IN_CYCLE corresponds to the general category. This means that the reward can be earned at any step in the game. While the auxilliary reward MOVED_TO_CRATE can only be earned when a crate or an opponent is present in the features, thus its category is Crates, Opponents.

Category	Auxilliary Rewards	Value
General	e.VALID_ACTION	0
	NOT_VALID_ACTION	-2000
	MOVED_IN_CYCLE	-2000
	NOT_MOVED_IN_CYCLE	0
Coins	e.COIN_COLLECTED	1000
	e.COIN_FOUND	20
	MOVED_TO_COIN	50
	MOVED_AWAY_FROM_COIN	-5
Crates, Opponents	e.DIDNT_DROP_BOMB	-60
	MOVED_TO_CRATE	100
	MOVED_AWAY_FROM_CRATE	-50
Bombs	MOVED_TO_BOMB	-2000
	MOVED_AWAY_FROM_BOMB	200
	MEANINGFUL_WAIT	5
	NOT_MEANINGFUL_WAIT	-30
	SAFE_FROM_BOMB	200
	NOT_SAFE_FROM_BOMB	-200
	MEANINGFUL_BOMB_DROP	200
	NOT_MEANINGFUL_BOMB_DROP	-400
Life	e.KILLED_SELF	-2000
	NOT_KILLED_SELF	0
	e.KILLED_OPPONENT	1000
	e.OPPONENT_ELIMINATES	50
	e.GOT_KILLED	-2000
	e.SURVIVER_ROUND	0

Table 1: Auxilliary rewards and their values

3.1.1.1 General Sven Zelch

At first we only had the rewards e.VALID_ACTION and NOT_VALID_ACTION and given them the same reward/punishment. For example e.VALID_ACTION = 10 and NOT_VALID_ACTION = -10. On the one hand we found out that rewarding him to much for a valid action would just result in him doing random valid moves. On the other doing an invalid action is really bad for the agent since it results in a lot of waiting, because off the agent trying to run into walls. That is why we reduced the reward for valid actions and increased the punishment for invalid action a lot. During this process we also found out, that the agent often gets stuck in loops, e.g. moves up and down 20 times in a row. To address this issue we implemented a MOVED_IN_CYCLE punishment and a corresponding NOT_MOVED_IN_CYCLE reward. If the new state is within the last two states the agent moved in a small cycle. With this we only

fixed the issue of small cycles like the up and down example, because we only look at the last two steps. However large cycles like walking around a one block wall happened very rarely and we did not concern this as an issue.

3.1.1.2 Coins Sven Zelch

The two rewards e.COIN_COLLECTED and e.COIN_FOUND were already implemented. We first decided to give both a high reward, because finding and collecting coins are crucial to scoring points and by that winning the game. After some testing we saw strange behavior from our agent. He blew up crates and found coins, but never seemed to bother picking them up. Therefore we designed MOVED_TO_COIN, which rewards the agent when it is moving to the nearest coin. We also lowered the reward for finding a coin, because it did not seem to important, because the agent already gets rewarded from blowing up crates and now also moving to the coin if one was hidden under a crate. MOVED_AWAY_FROM_COIN was also created since it was easy to implement with MOVED_TO_COIN already created, but we found out that punishing the agent to much for moving away from a coin results in the agent paying less attention to bombs by blindly walking to coins.

3.1.1.3 Crates, Opponents Sven Zelch

Destroying crates was not important at first, because the agent was playing on the coin-heaven scenario. When it was playing alone on the classic scenario it just tried to collect coins and did not blow up crates. Which means that quite often the nearest coin was not reachable and the agent was constantly running into a wall. To solve this we rewarded the agent for destroying a crate with the e.CRATE_DESTROYED reward. Since this seemed to have no effect, because it took the agent to long to learn it, we also added a DIDNT_DROP_BOMB punishment, if the agent is in front of a bomb and does not drop a bomb. That still did not show great results so we added the two opposite rewards MOVED_TO_CRATE and MOVED_AWAY_FROM_CRATE which finally showed some promising results when it comes to him destroying crates, but he often tends to kill himself.

3.1.1.4 Bombs Sven Zelch

Because our agent often killed himself we punished it highly when he moved to a bomb. MOVED_TO_BOMB punishes the agent when it walks into bomb explosion range and MOVED_AWAY_FROM_BOMB rewards it if it moves out of the range. Additionally we punished the agent when it is in a position, where the bomb explosion would kill it and rewarded it when it is in a position where the bomb does not kill him. We also saw the the agent often was waiting in bomb range. Therefore we added MEANINGFUL_WAIT and NOT_MEANINGFUL_WAIT which rewards it if it waits in a save spot and punishes it if it waits in bomb explosion range. Also the agent placed a lot of unnecessary bombs. That is why we added MEANINGFUL_BOMB_DROP and NOT_MEANINGFUL_BOMB_DROP, which rewards the agent if the bomb placed would destroy a crate and punishes it otherwise.

3.1.1.5 Life Sven Zelch

Obviously getting killed is bad for the agents score, because it can not increase it anymore while other agents can. We added a hard punishment for suicides e.KILLED_SELF and getting killed e.GOT_KILLED. On the other hand killing a opponent is great for winning a game so we rewarded e.KILLED_OPPONENT highly. Over the course of designing rewards we really liked opposite rewards that is why we also added NOT_KILLED_SELF and e.SURVIVED_ROUND but in this case they don seem to be good, because more specific rewards are better, e.g. MOVED_AWAY_FROM_BOMB for moving out of bomb range, and they have barley any impact.

3.1.2 Reward shaping Angelina Basova

In addition to auxilliary rewards we also used a dynamic potential-based reward function F. The goal was to incentivize the agent to end the round fast, without meaningless moves. Therefore, we defined the function F as the negative time step of the previous state. Formally:

$$F(s, t, s', t') = -1 * step_s$$

where s and t is the previous state and its time step and s' is the new state arrived at time step t'. As the game settings define a maximum

of 400 time steps and a minimum of 0 time steps, the function F is bounded with $-400 \leq F \leq 0$. The agent receives a total reward R,

$$R = r + F(s, t, s', t')$$

where r is the sum of the auxilliary rewards and F is the dynamic potential-based reward function.

4 Experiments and Results

In the following section we report Experiments and their results for our two agents. We begin to analyse the Q-Learning implementation. Then we analyse the deep Q-Learning implementation.

4.1 Q-Learning

In this subsection we analyse our Q-Learning agent. First we illustrate the values of our final Q-Table. Second, we compare our agent's result in the classic and coin-heaven scenario. Finally, we test our agent's performance with the four tasks on the exercise sheet.

4.1.1 Final Q Table values Angelina Basova

The agent showed some promising results in specific circumstances. The following points discuss some problems and solutions to improve the agent.

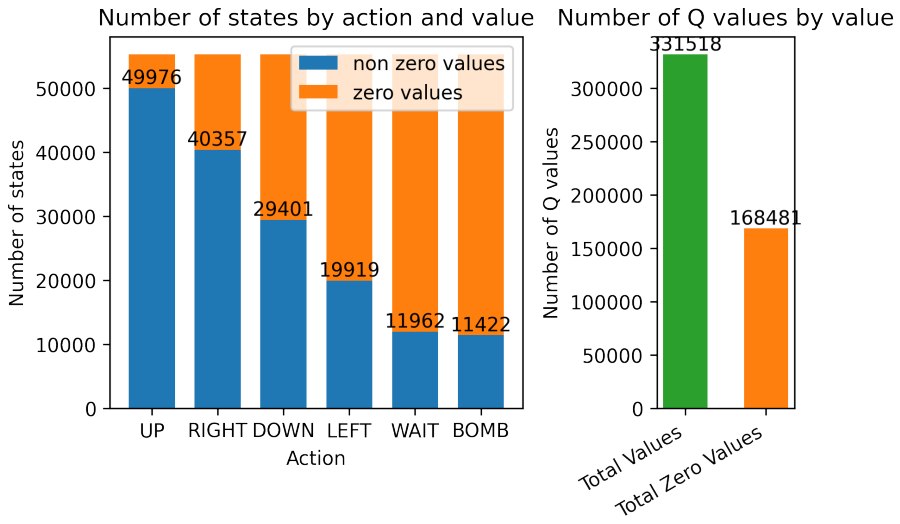


Figure 2: Overview of zero values in the Q table. The left Figure shows the ratio of zero to non - zero values per action. The right Figure shows the overall zero to non-zero ratio in the entire Q table.

Our final Q table consists of 55253 states and 331518 Q values. A major bottleneck in the development of the agent was the lengthy training. The majority of the last week before the submission was utilized for training. However, this proved to not be enough as the right chart of Figure 2 shows. About half of the Q table consists of zero values, which means that half of the actions were never tried out. However, the distribution of zero values is not equal among the six actions. As the left chart of Figure 2 shows, the majority of the zero values are found in the actions WAIT and BOMB. Specifically, these actions were executed only in 20% of the states. In contrast, the action UP was executed in 49976 out of the 55253 states, or in other words in over 90% of the states.

This ratio aligns with our implementation during exploration. As stated in Section **??**, during exploration the first zero valued action was selected. Therefore, whenever a new state was found, the first selected action was always UP. While to select the action BOMB a state should be visited for six times during exploration, or should be selected randomly when $e \neq 1$

4.1.2 Scenario-related performance comparison Angelina Basova

The submitted agent exceeds the majority of the think time due to an iteration while searching for similar states. As the exceeding in think time isn't related to Q-Learning but to the programmatic implementation of a search function, we perform the following experiments after replacing the iteration with a scipy function. The result of the search function remains the same. However, this way we are able to illustrate the performance of our Q-Learning implementation considering the selected hyperparameters and auxilliary rewards.

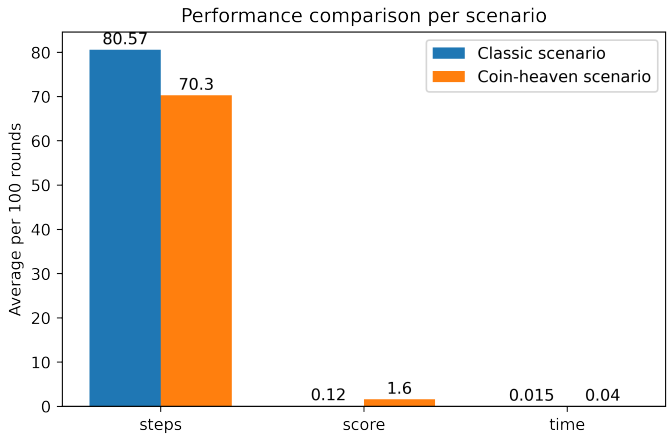


Figure 3: Comparison of game metrics in the classic and coin-heaven scenario. The values correspond to an average value per round after playing for 100 rounds.

Figure 3 shows the game metrics in the classic and the coin-heaven scenario. Each value equals to the average value per round after playing 100 rounds.

In the classic scenario, the agent plays on average for 80 steps. In contrast, in the coin-heaven scenario the agent plays only for 70 steps. In both scenarios, the average round ends way before the 400 steps are achieved. The reason lies at that the use of a dynamic reward shaping function in combination with a high γ hyperparameter. A high γ emphasizes future rewards, which incentivizes the agent to collect a high cumulative ?? reward over the round. However, the dynamic reward shaping function reduces the reward by the time step,

to incentivize the agent to finish the round fast. Therefore, the agent is incentivized to end the round early to maximize its rewards, which is clearly the case in both scenarios.

In the coin-heaven scenario, the average score equals to 1.6 points which is by 13 times higher. While in the classic scenario, the agents earns 0.12 points. As the agent was primarily trained in the coin-heaven scenario, the Q values of the states of the coin-heaven scenario converged better than the Q values of the classic scenario.

The think time in the coin-heaven scenario is by 2.5 times higher than in the classic scenario. It is unclear why the think time is higher. We expected to observe higher think time in the classic scenario as this scenario was less explored during training.

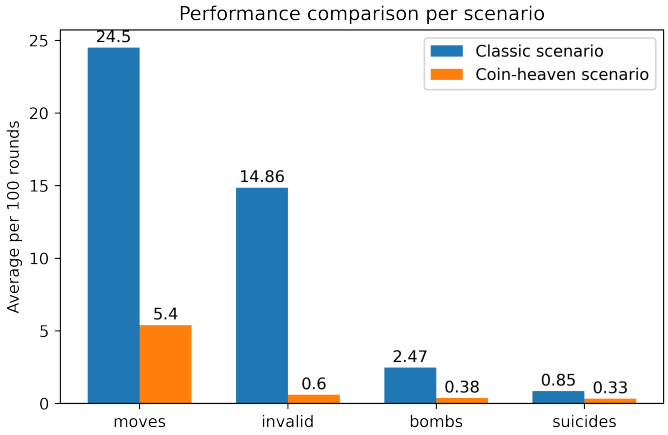


Figure 4: Comparison of performance in the classic and coin-heaven scenario. The values correspond to an average value per round after playing for 100 rounds.

Figure 4 shows the effect of the moves in the classic and the coin-heaven scenario.

Over all four metrics the classic scenario contains consistently higher values. This indicates the worst performance in comparison to the coin-heaven scenario, which aligns with our focus on training in the coin-heaven scenario. This observation becomes even more obvious when considering bad actions such as invalid actions and suicides. In the coin-heaven scenario the agent chooses on average 24 times more invalid actions and performs 2.5 times more suicides.

In the classic scenario, as the agent is less cautious with its moves, its performs on average 24.5 moves per round. This equals to 5 times more moves than in the coin-heaven scenario.

Surprisingly, the agent places 6 times more bombs in the classic scenario. Despite the shorter training time, the beneficial action is selected way more times than in the coin-heaven scenario. Given that there are less crates and opponents in the coin-heaven scenario than in the classic scenario, the auxilliary rewards incentivized the use of bombs in the classic scenario. While 2.4 bombs are certainly not enough to win the round, the bomb ratio between the two scenarios shows that the rewards give the right incentives. The main issue remains the long training time until convergence.

Overall, the agent shows better performance in the coin-heaven scenario than in the classic scenario. The performance is consistently better over all metrics except the time. The longer training time resulted into more Q values closer to convergence. Nevertheless, we observe good agent behavior in both scenarios, such as a low number of invalid actions and more bomb placing in the scenario with crates and opponents.

4.1.3 Task-related performance comparison

4.1.3.1 Task 1 Sven Zelch

In Task 1 an agent is supposed to collect all the coins in the coin-heaven scenario, where no other agents and no crates exists.

Figure 5 compares the performance of our agent with the coin_collector_agent for Task 1. The coin_collector_agent shows an ideal scenario of how our agent should behave he reaches a score of 50 in all 100 games played, which means he always collects all 50 coins in the coin-heaven scenario, because it is the only way he can increase his score and each collected coin increases its score by one. Our agent struggles to collect the coins within the given time of 400 steps. In fact he only collects 1 coin every 10 games. Watching the game with GUI on we can see the agent is waiting a lot. This issue occurred before where the time it takes for the agent to chose an action was greater than 0.5 seconds and the agent was forced by the game to wait. We are unsure what causes this issue this time around.

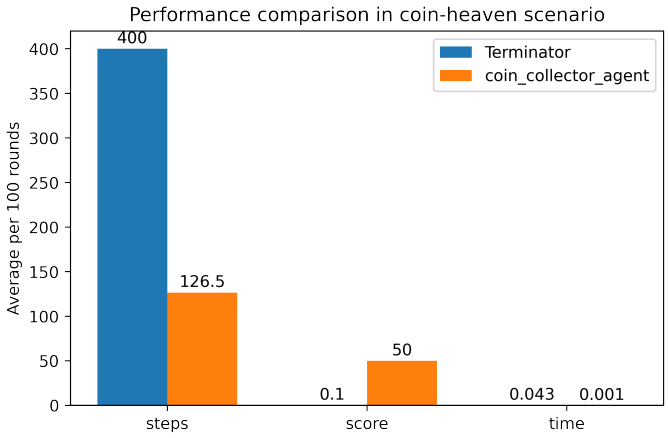
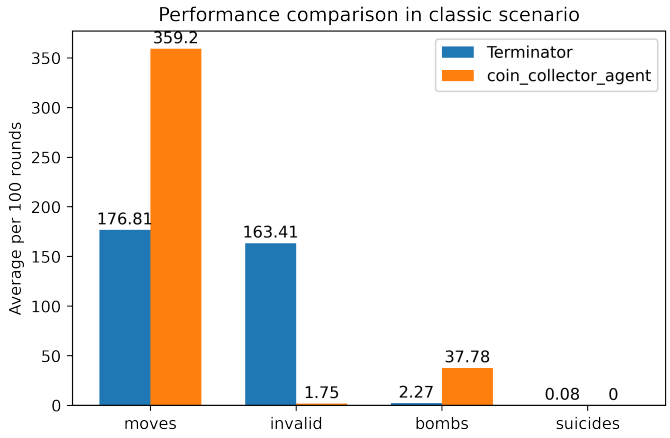


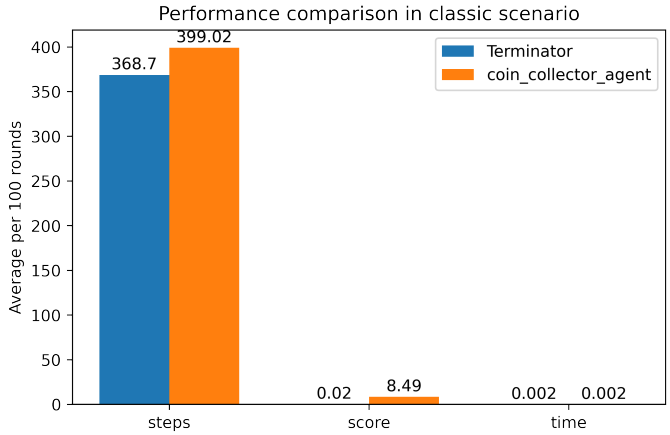
Figure 5: Performance comparison Terminator and coin_collector_agent in coin-heaven scenario.

4.1.3.2 Task 2 Sven Zelch

Task 2 is similar to Task 1, but this time some coins are hidden under crates. The crates need to be blown up first before the agent can collects the coins. We used the classic scenario without other agents for this task.



(a) Performance comparison Terminator and coin_collector_agent moves



(b) Performance comparison Terminator and coin_collector_agent game

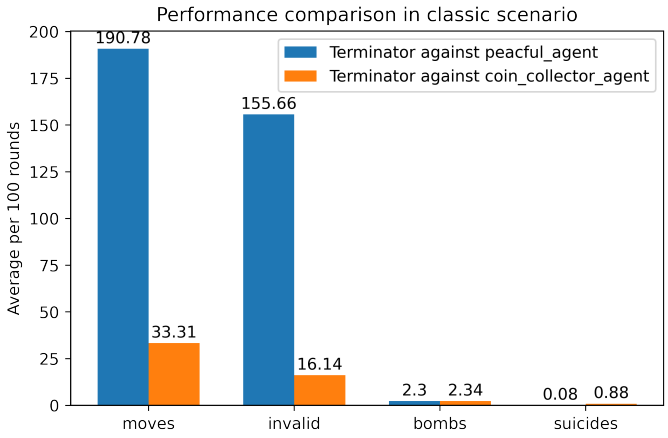
Figure 6: Performance comparison Terminator and coin_collector_agent in classic scenario without opponents

Figure 6a and 6b compare the performance of our agent with the coin_collector_agent for Task 2. Again the coin_collector_agent performs way better than the Terminator, especially score wise. Also it does about two times more moves than our agent and if we only count valid moves: $(359.2-1.75)/(176.81-163.41) \approx 27$ times more valid moves. The issue in Task 1 could also be caused by a lot of invalid actions, which look like the agent is waiting actions from the GUI perspective. As well as the limited amount of bombs Terminator drops

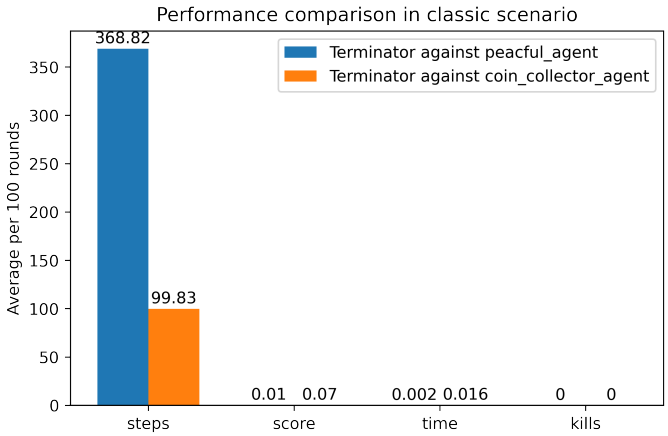
is caused by the limited small amount of valid actions, about 13 per round, it takes. A good thing about our agent is that he only suicides 8 times in 100 games which is worse than the coin_collector_agent, but still a great result nevertheless. Both agents seem to struggle to find all the coins in 400 steps. Terminator has a lower step count, because he suicides himself sometimes. Something really interesting here is that both agents take approximately the same time to chose an action. Particularly comparing the result from Terminator with them from Figure 5 a high reduce in time to chose an action can be seen. This is probably caused by the agent being mostly trained in the classic scenario.

4.1.3.3 Task 3 Sven Zelch

In this Task the agent should additionally to collecting coins also try to hunt and blow up an opposite agent.



(a) Performance comparison Terminator against peaceful_agent and Terminator against coin_collector_agent moves



(b) Performance comparison Terminator against peaceful_agent and Terminator against coin_collector_agent game

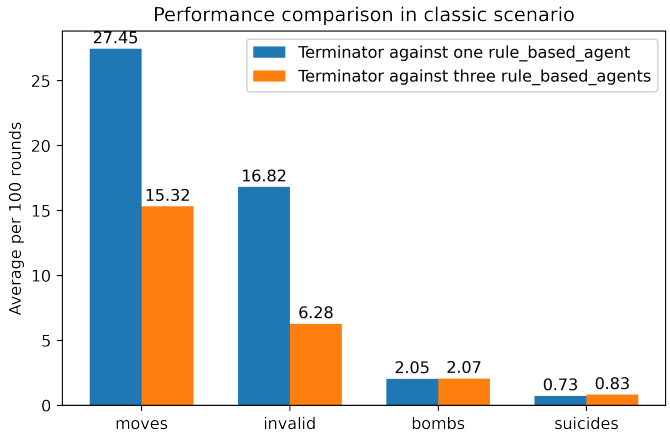
Figure 7: Performance comparison Terminator against peaceful_agent and Terminator against coin_collector_agent

Figure 7a and 7b compare the performance of our agent Terminator against the peaceful_agent, who does not drop bombs and against the coin_collector_agent who only drops bombs to destroy crates. The peaceful_agent seems to have barley any influence on the agent, because results of the match against the peaceful against are similar to the results of Figure 6a and 6b. Interestingly however is that the

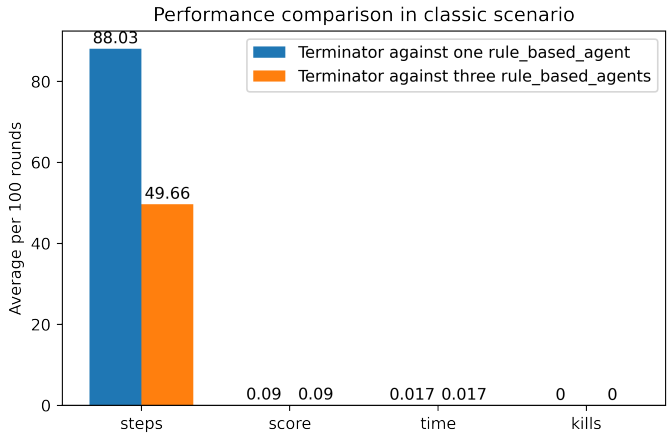
coin_collector_agent has a drastic effect on Terminators performance. The suicide rate is 11 times higher, even tough he drops the same amount of bombs. As a result of that he performs less moves, but about half his actions are valid actions compared to a fifth. Regardless of less rounds played the score is a bit better, but that is probably caused by small sample size. In both cases Terminator failed to kill his opponent.

4.1.3.4 Task 4 Sven Zelch

The agent competes against one ore more agents for the highest score in a classic Bomberman game.



(a) Performance comparison Terminator against one rule_based_agent and three against rule_based_agents moves



(b) Performance comparison Terminator against one rule_based_agent and three against rule_based_agents game

Figure 8: Performance comparison Terminator against one rule_based_agent and three against rule_based_agents

Figure 7a and 7b compare the performance of Terminator against one rule_based_agent and against three rule_based_agent. The rule_based_agent not only tries to collect coins but also to kill other opponents. Terminator has a hard time surviving against the rule_based_agent. With three rule_based_agent in the game he dies significantly faster than with only one. Interestingly the score is the same in both cases. This

might be caused by the agent collecting a coin before an other agent can even kill him. Other than that he still did not manage to kill an opponent.

5 Conclusions

Q-Learning Sven Zelch Angelina Basova

Overall, we developed a Q-Learning agent with substantially better performance in a coin-heaven scenario than in the classic scenario. Our agent performs more suicides when playing with other agents. However, on its own the agent collects more points and survives more time steps. However, the agent has the potential to converge to the optimal policy as our implementation satisfies the convergence conditions. Specifically, because the *epsilon* value decreases over time and the rewards are bounded.

Our Q-learning agent had a major issue which only happened after training him for a long time and caused him to take more than 0.5 seconds to chose an action. The issue was easily fixed after the deadline. Next time we will utilise more time for testing our agent. Despite that, we recommend spending a lot of time carefully designing rewards, which have a great influence on the performance improvement of the agent. Substantial improvements were seen by designing opposite rewards.

A possible improvement of the agent could start with a different state representation. For instance, the four elements related to the environment could be reduced to one elements stating the best action. This way, through the smaller feature space, the Q-table would converge faster.

The game setup could be improved by adding a 9x9 version off the game and its game modes. Improvements of the agent can be seen much faster, particularly when testing rewards.

References

[1] Sam Devlin and Daniel Kudenko. “Dynamic Potential-Based Reward Shaping”. In: *Proceedings of the 11th International Con-*

ference on Autonomous Agents and Multiagent Systems (AAMAS 2012). Valencia, Spain: International Foundation for Autonomous Agents and Multiagent Systems, 2012, pp. 433–440. ISBN: 0981738117.

[2] Junqin Jiang et al. "Policy invariance under reward transformations: Theory and application to reward shaping". In: *Kao Neng Wu Li Yu Ho Wu Li/High Energy Physics and Nuclear Physics* 23.12 (1999), pp. 1156–1157. ISSN: 02543052.

[3] M.L. Littman. *Markov Decision Processes*. Cambridge, MA, USA: Elsevier, 2001, pp. 9240–9242. ISBN: 0262039249. DOI: 10.1016/B0-08-043076-7/00614-8. URL: https://linkinghub.elsevier.com/retrieve/pii/B0080430767006148.

[4] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018. ISBN: 0262039249.

[5] Christopher J. C. H. Watkins and Peter Dayan. "Q-learning". In: *Machine Learning* 8.3-4 (1992), pp. 279–292. ISSN: 0885-6125. DOI: 10.1007/BF00992698. URL: http://link.springer.com/10.1007/BF00992698.