Heidelberg University

End of the Semester Project

Lecture: Machine Learning Essentials

Report

Using Reinforcement Learning Methods to Train an Agent to Play Bomberman

Institute:	Computer Vision and Learning Lab Biomedical Genomics Group
Authors:	Leonie Boland, MatNr. 4055040 Kevin Klein, MatNr. Berkay Günes, MatNr.
Version from:	September 29, 2023
Supervisor:	Prof. Dr. Ullrich Köthe

1 Declaration

Contents

2 Introduction

The integration of artificial intelligence (AI) into the realm of video gaming has led to groundbreaking advancements in gameplay and immersive experiences. One of the most intriguing challenges in this endeavor is the development of AI agents capable of mastering complex and dynamic games such as Bomberman. Bomberman, a classic arcade-style game, presents a rich and multifaceted environment with intricate decision-making, spatial reasoning, and strategic planning. Solving Bomberman with an AI agent using reinforcement learning techniques is a fascinating and intricate problem, one that holds significant promise for the AI and gaming communities.

Reinforcement learning (RL), a subfield of machine learning, offers a compelling approach to tackle the Bomberman challenge. RL revolves around training agents to learn optimal strategies by interacting with their environment, taking actions, and receiving feedback in the form of rewards or penalties. In the context of Bomberman, RL techniques can be instrumental in enabling AI agents to navigate mazes, strategically place bombs, avoid traps, and outsmart opponents. Here are a few noteworthy reinforcement learning techniques that hold potential in solving the Bomberman game:

Q-Learning: Q-learning is a classic RL algorithm that could be applied to Bomberman. It enables agents to learn a value function that maps state-action pairs to expected cumulative rewards. By exploring different actions in the game and updating Q-values, an AI agent can eventually converge on an optimal strategy.

Deep Q-Networks (DQN): DQN extends Q-learning by employing deep neural networks to approximate the Q-value function. This technique has been successful in handling complex and high-dimensional state spaces, making it a strong candidate for Bomberman.

Policy Gradient Methods: Policy gradient methods aim to directly learn a policy that specifies the agent's actions in different game states. This approach can be effective in scenarios where the optimal policy is not easily represented by a value function, as it allows for a more flexible and direct mapping from states to actions.

Proximal Policy Optimization (PPO): PPO is a state-of-the-art RL algorithm that focuses on optimizing policy functions. It offers stability and strong performance in challenging environments, making it suitable for Bomberman's dynamic and adversarial setting.

Actor-Critic Methods: Actor-critic methods combine value-based and policy-based approaches, leveraging both a critic (value function) and an actor (policy). This combination can provide the agent with a more robust learning framework, enhancing its decision-making capabilities.

In the pursuit of solving Bomberman with an AI agent, the selection and fine-tuning of the appropriate RL technique, as well as the integration of domain-specific features, will play pivotal roles in achieving success. This endeavor not only serves as a compelling testbed for reinforcement learning but also has the potential to elevate the overall gaming experience by delivering AI opponents that are challenging, adaptive, and engaging.

3 Methods

In this section, we elucidate our methodology and approaches concerning the various machine learning models that we have developed for the Bomberman game agent. We will also substantiate what has proven effective, articulate the concepts we have discarded, and expound upon the rationale for ultimately selecting the model we have submitted.

3.1 Our final best model (first project)

Explained by Berkay

3.2 Our second best model (second project)

In our quest to develop our second best model for Bomberman, we embarked on a journey where we collected data pairs of game states and rule-based agent actions. Using this data, we trained a machine learning model, optimizing it to mimic the rule-based agent's decision-making. The AI, powered by a carefully chosen model architecture and loss function, should learn to imitate the agent's actions in various game states. This approach allowed us to create an AI actor that could navigate Bomberman's challenges and make decisions based on the learned rules and actions of the rule-based agent.

To enhance our model and surpass the rule-based agent's performance we want to use deep Q-learning. We extended our actor-critic architecture by introducing a critic network to estimate Q-values, allowing the agent to understand the long-term consequences of its actions. We implemented an exploration strategy, such as epsilon-greedy, to encourage the model to explore new strategies and avoid getting stuck in suboptimal actions. Lastly, we employed a reward shaping mechanism that provided tailored rewards for encouraging desirable behaviors.

The basic idea is as follows: we first pretrain a model that imitates the rule-based agent and then further improve this model using deep Q-learning. However, the deep Q-learning part is not explained in detail in this section since the architecture is almost identical to what was explained in the section about our final project. Nevertheless, we do discuss where there were changes, such as in the features.

We chose PyTorch as our primary library because it offers a flexible and dynamic computation graph, allowing us to easily design and experiment with complex neural network architectures. Additionally, its extensive community support, rich ecosystem of pre-built modules, and integration with GPU acceleration make it an ideal choice for achieving our goals efficiently.

Data Collection

In our data collection process, we began by generating extensive game data through multiple playthroughs of Bomberman with our rule-based agent. During these sessions, we recorded both the game states and the corresponding actions taken by the agent, encompassing movements (left, right, up, down) and bomb placements.

The code below explains how we implemented it. First, we create a ReplayBuffer from the rule-based agent to record how the rule-based agent played. The buffer returns a tuple containing the list of states and actions taken by the rule-based agent when

`get` is called. The actions are one-hot encoded so that they can later be used as classes during network training. In `setup_training`, the `RuleBased_Replay` is initialized, where `capacity` determines how many games we want to store, typically ranging from 500,000 to 1,000,000 games. In `game_events_occurred`, we store each state along with the corresponding action taken by the rule-based agent because we have the rule-based agent play only for training purposes in `callback.py`.

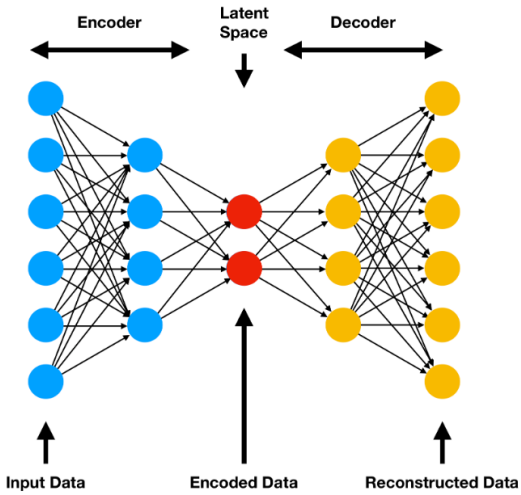
```
    ...

    from collections import namedtuple, deque

    ...

    class RuleBased_Replay():
        def __init__(self, capacity):
            self.states = deque([], maxlen=capacity)
            self.one_hot_labels = deque([], maxlen=capacity)

        def push(self, state, action):
            self.states.append(state)

            # convert action to one-hot
            a = [0,0,0,0,0,0]
            a[action] = 1
            self.one_hot_labels.append(a)

        def get(self):
            return self.one_hot_labels, self.states

        def __len__(self):
            return len(self.one_hot_labels)

    ...

    def setup_training(self):

    ...

        self.rule_based_replay = RuleBased_Replay(capacity=MEMORY_BUFFER)

    ...

    def game_events_occurred(self, old_game_state: dict, self_action: str,
    new_game_state: dict, events: List[str]):
        self.rule_based_replay.push(
            state_to_features(old_game_state),
            ACTIONS.index(self_action))

        ...

    ...
```

Once we amassed a substantial dataset, we transformed it into a format compatible with PyTorch's Dataset class. Each dataset entry consisted of two fundamental components: the game state and the associated action label. We represented the game states as tensors to ensure seamless integration with PyTorch.

```
    ...

    train_ds = CTDataset(data, labels)

    ...
```

Leveraging PyTorch's Dataset class, we crafted a custom dataset equipped with efficient data loading and preprocessing capabilities. During training, we harnessed PyTorch's DataLoader to batch and load the data conveniently. Depending on the problem at hand, we also applied data augmentation techniques, such as random rotations or flips, to enhance the diversity of our training samples and boost the model's generalization capabilities.

```
    ...

    train_dl = DataLoader(train_ds, batch_size=batch_size, shuffle=True)

    ...
```



(a)

```
 0  0  0  0  1  0  1
 0 -1  0 -1  1 -1  1
 0  0  0  0  0  0  1
 0 -1  0 -1  1 -1  0
 0  1  0  0  1  0  1
 0 -1  0 -1  1 -1  0
 0  0  0  0  0  0  0
```

(b)

Figure 1: a: Example of an autoencoder network [1]. b: The 7x7 game state is getting more and more sparse over time.

To facilitate model training and evaluation, we divided the dataset into distinct sets for training, validation, and testing purposes. This partitioning allowed us to gauge our model's performance on unseen data and make necessary adjustments.

State Representation

In our past work on Bomberman, we recognized the critical importance of state representation in optimizing the performance of our AI agents. Specifically, we focused on the transition from the original 17x17 grid, which encompassed the entire Bomberman field, to a reduced 7x7 grid. To accomplish this, we centered the agent's viewpoint in the 17x17 grid to capture the immediate surroundings. This reduction was not just a matter of simplification, it is especially important to prevent overfitting. When the agent network consistently receives the entire 17x17 field for learning, there's a high likelihood that the network will memorize training patterns and struggle with test data. The objective is to ensure that the agent learns the intelligent strategies of the rule-based agent to a certain extent, at the very least, mastering how to play the game effectively without necessarily striving for extreme efficiency. In doing so, we also took inspiration from humans who, like our AI, do not perceive the entire field but rather focus on a specific part while filtering out the rest. Reducing the field to 7x7 retained key spatial relationships and relevant game features while simplifying the representation, thus allowing our AI agents to focus on essential information. To determine if there is a difference in how the rule-based agent plays the game when the view box is reduced to 7x7, we had the following idea: lets consider the following function:

$$G_A(f, v) \rightarrow \{\text{up, down, left, right, bomb}\} \tag{1}$$

that outputs the best possible action the agent could do. The field is described by f and v is the viewbox of the agent. We did many test rounds with the rulbased agent on an 17x17 field with `--seed 1` for a comparable result. We had two scenarios in one scenario the rulbased agent saw the whole 17x17 field e.g $G_a(f, 17 \times 17)$, on the other scenario the agent saw a 7x7 field e.g $G_a(f, 7 \times 7)$. Then we calculated the probability p to find out how likely it is, that the actions in both scenarios are different in one step.

We simply counted the number of different actions and divided them by the maximum number of steps. For the 7x7 field we had a probability of $p = 0.054$ which was the best value for us compared to a 5x5 or 3x3 viewbox.

But, please note that initially, we interpreted this number p quite differently. The probability distribution is not uniform. It turns out that the agent with a 7x7 viewbox makes more diverse actions as the game progresses compared to the agent with a 17x17 viewbox. This happens because towards the end of the game, there are fewer items or opponents within the 7x7 view, and the agent becomes uncertain about its actions. A better approach would be to determine the probability distribution function $p(X = k)$, where k represents the number of steps in a game. For example lets look at the probability at the beginning of the game $p(0 \geq X \leq 20) = 0$ and at the end of the game $p(310 \geq X \leq 330) = 0.47$ We had to come up with a solution to address this issue. Once the 7x7 field was empty of items or opponents, the agent switched to an exploration mode and randomly searched the surroundings for potential objectives.

However, the 7x7 field often contained sparse feature data see Figure 1b, with many cells remaining empty or irrelevant. To efficiently handle this challenge, we employed an encoder-decoder network see Figure 1a. This architecture allowed us to compress the sparse feature data effectively. But before we delve into how the encoder-decoder network functions, we need to address how our features are generated within the `state_to_features` function. First, we restrict the agent's field of view to 7x7. Then, we create a 7x7 map with the agent in the center, represented by the number 100. Here, 0 denotes open space, 1 represents crates, -1 signifies walls, 5 indicates coins, 15 represents opponents, and 25 stands for bombs. Another 7x7 map contains the hidden features, which are features that would otherwise overlap with those on the first map. These hidden features include whether all agents can ignite a bomb, represented as 0 or 1. A third map with hidden features describes the blast radius of a bomb along with its respective countdown. These three 7x7 maps, also known as channels, are then merged into a single 147-element array. Below is a code snippet:

```
    ...

    def state_to_features(game_state: dict) -> np.array:
        field = game_state["field"]
        bombs = game_state["bombs"]
        explosion_map = game_state["explosion_map"]
        coins = game_state["coins"]
        agent = game_state["self"]
        others = game_state["others"]

        my_pos = make_field(agent[3])

        ...

        # non hidden features
        # make_field create a matrix from coordinate list
        channel1 = make_field(bombs) + make_field(others) + make_field(coins)
                                     + field + my_pos

        # can agent put a bomb
        channel2 = extract_bom_is_possible_to_field(others) +
                   extract_bom_is_possible_to_field(agent)

        channel3 = make_real_explosion_map(bombs, explosion_map)

        ...

        vf = make_viewbox(7,7, [channel1, channel2, channel3])
        return vf.flatten()

    ...
```

Next, we want to modify the `state_to_features` function to further compress the data in the 149-element array while preserving as much information as possible. To achieve this, we'll utilize an encoder-decoder network see Figure 1a. This network has the same output dimension as the input dimension and aims to reconstruct the input data. However, in the dense layers, the features are reduced up to a certain point (encoded data layer). Then, in subsequent dense layers, the features are expanded back to the input dimension. If the network can effectively reconstruct the input data, we'll use it and extract the encoded data layer, which we will return in the `state_to_features` function. We were able to compress the data from the 149-element array to a 72-element array. These data will now be used for our second network to train the rule-based agent. Details about both networks will be explained later in Model Architecture. Here's the modification of the `state_to_features` function once again:

```
...

def state_to_features(game_state: dict) -> np.array:

    ...

    vf = make_viewbox(7,7, [channel1, channel2, channel3])
    compressed_features = encoder_decoder_network.encode(vf.flatten(), 72)
    return compressed_features

...
```

The encoder module was instrumental in feature learning, condensing the sparse data to capture essential features efficiently. Additionally, it reduced dimensionality, enhancing computational efficiency and reducing noise in the data. During decoding, the network could reconstruct the original 7x7 state representation from the compressed data, ensuring that no critical information was lost in the process.

Model Architecture

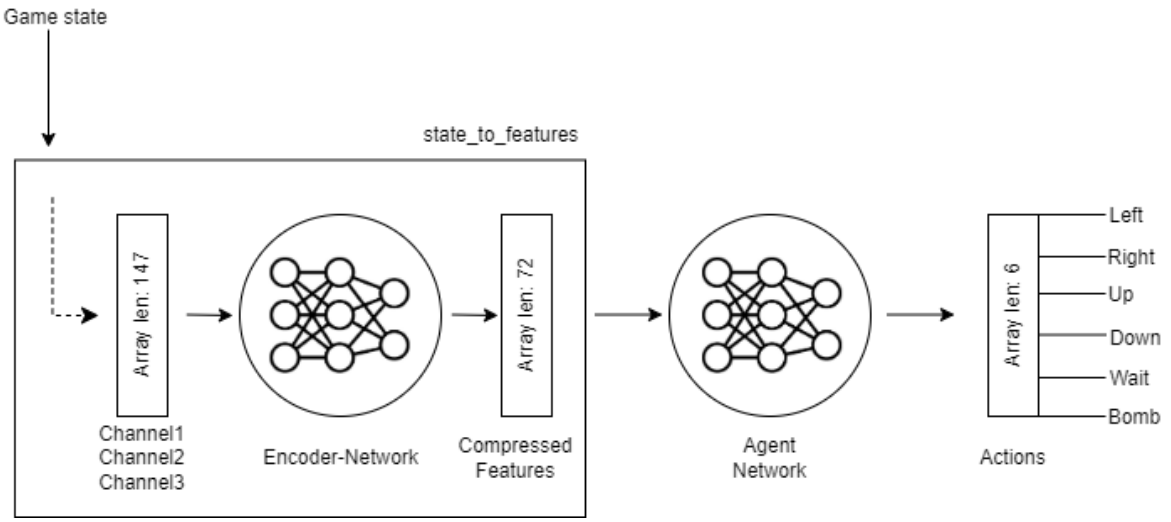


Figure 2: Here is how our two models come into play.

In total, we employ two neural networks see Figure 2. The encoder network is utilized within the `state_to_features` function to compress the generated features, as they contain many repetitive entries, such as 0, 1, or -1. Through this compression, our aim is not only to reduce the data size but also to create patterns that make it easier to

describe a state. This is particularly significant for the second network. The second network is the agent network, which receives data from the `state_to_features` function and endeavors to predict the best action. Here are the characteristics of the two networks listed:

```
Encoder Network:
    Layers: 1 Inupt, 5 Hidden, 1 Output
    Neurons:
        Input1: 147, Hidden1: 124, Hidden2: 102, Hidden3: 72,
        Hidden4: 102, Hidden5: 124, Output1: 147

Agent Network:
    Layers: 1 Inupt, 3 Hidden, 1 Output
        Neurons:
            Input1: 72, Hidden1: 64, Hidden2: 32, Hidden3: 16, Output1: 6
```

Loss function and optimizer

We have two models: the encoder network for further feature compression and the agent network, which is tasked with learning how to play the game effectively. Additionally, the agent is initially meant to imitate the rule-based agent, which is achieved through classification. Later on, the agent will be further improved using deep Q-learning. For all these scenarios, we require different loss functions and optimizers with varying learning rates because there are more suitable optimizers and loss functions for specific cases.

For the encoder network, the ADAM optimizer with a learning rate of 0.001 proved to be the most effective, and the chosen loss function was the MSE (Mean Squared Error) loss. During experimentation, we observed that the loss function did not converge to near zero with a learning rate of > 0.001, and with a learning rate of < 0.001, it converged towards zero, albeit very slowly.

For the agent network when mimic the rulbased agent, we used the ADAM optimizer with a learning rate of 0.0001 and the Cross-Entropy loss function because in classification, the goal is to assign probabilities to each class. Cross-entropy loss directly deals with probability distributions and can handle multi-class problems efficiently.

For the agent network during deep Q-learning, we used the ADAM optimizer with a learning rate of 0.0001 and the Huber loss as the loss function. It is designed to balance the benefits of mean squared error (MSE) and mean absolute error (MAE) loss functions. But for any reason was the MSE loss better for the encoder network than the Huber loss.

3.3 Other aproaches that we had

At the beginning of the project, we had several other ideas about how to construct our AI agent. One alternative idea was to use Q-tables. For the coin collector agent, we considered the concept that it would only perceive one coin, specifically the one closest to it. However, in the end, we discarded all of these ideas, and the following reasons explain why:

3.3.1 Q-Tables

Q-tables (Quality or Q-value tables) are used to approximate and store the expected cumulative rewards (Q-values) associated with different state-action pairs in a Markov Decision Process (MDP). Rows represent states or state descriptions, columns represent possible actions that can be taken in those states. Each cell in the table stores the expected cumulative reward, denoted as Q-value, for taking a specific action in a specific state.

The Q-value for a state-action pair (s, a) represents the expected sum of rewards an agent can achieve by taking action a in state s and following a specific policy thereafter. The Q-value is typically updated iteratively as the agent explores and learns from its interactions with the environment. Once the Q-table has converged or reached a sufficiently accurate representation of the optimal Q-values, the agent can select actions that maximize these Q-values

The equation for updating the Q-table in the context of Q-learning is as follows:

$$Q(s, a) = (1 - \alpha) \cdot Q(s, a) + \alpha \cdot [R(s, a) + \gamma \cdot \max_a Q(s', a)] \tag{2}$$

In this equation:

- $Q(s, a)$ represents the Q-value for a specific state-action pair (s, a).
- α (alpha) is the learning rate, determining the weight given to the new information when updating the Q-value. It's a value between 0 and 1.
- $R(s, a)$ is the immediate reward received after taking action 'a' in state 's'.
- γ (gamma) is the discount factor, which controls the importance of future rewards. It's also a value between 0 and 1.
- $Q(s', a)$ represents the Q-value of the next state 's' after taking action 'a', and a is the action that maximizes this Q-value.

The rason why we discarded this idea is that the bomberman environment has a large number of states, even when using a reduced state representation. Each cell on the game board can have multiple attributes (e.g., walls, crates, coins, enemies, bombs), resulting in an extensive state space. Creating a Q-table to store Q-values for every state-action pair in such a high-dimensional space would be impractical.

3.3.2 Coin-Collector Agent that sees only one coin

Our coin collector agent was given the entire 17x17 field but only focused on the nearest coin, which was simply added to the `game_state["field"]`. It received rewards only when collecting this specific coin. This approach worked very well for us, and the agent could consistently collect nearly all the coins later in the game. The advantage of this method is that it doesn't matter how many coins are on the field; the agent doesn't need to be retrained. However, for an agent tasked with more complex actions like defeating enemies, destroying crates, and collecting bombs, we couldn't pursue this idea. This is because we explored other concepts to enable the agent to perform various tasks effectively, and these concepts are described in this report.

4 Training

4.1 second best agent

In this section, we describe how we trained our second-best agent. This agent initially imitates the rule-based agent and then seeks further improvement through deep Q-learning.

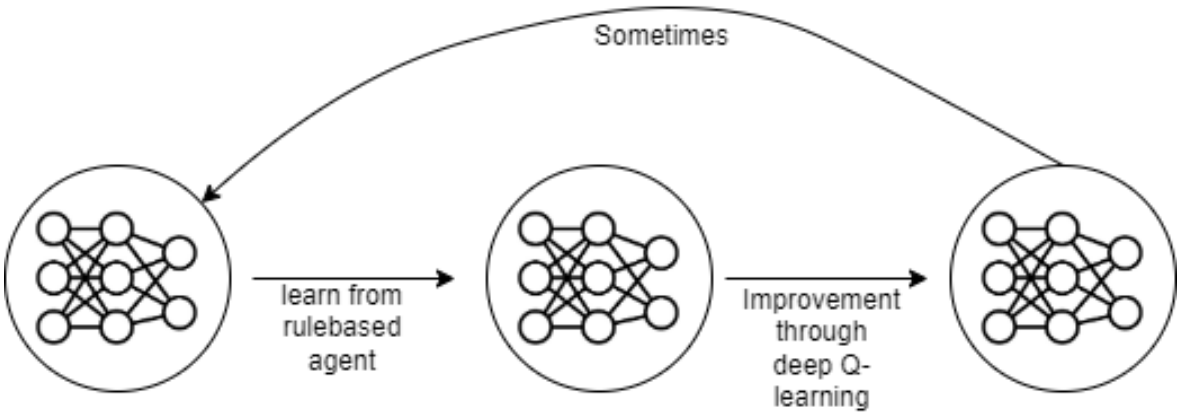


Figure 3: Here is how we trained our second best model.

Our initial idea was to develop an agent that starts by performing at the same level as the rule-based agent and then improves further through deep Q-learning. For this purpose, we had devised the following concept see Figure 3.

Initially, our agent attempted to imitate the rule-based agent, as mentioned earlier. This was achieved through a classification approach. We collected the actions of the rule-based agent for each game state, and these actions served as the labels for the respective states. We gathered between 500,000 and 1,000,000 action-label pairs in this manner. New data was added to the ReplayBuffer after each step until reaching the limit of 500,000 to 1,000,000 entries. After each round, specifically at the `end_of_round` event, the following simplified function was executed to train our model. Of course, the model was then also saved in an external file, which was achieved using Pickle.

```
    ...

    def learn_rulebased_agent(model, labels, data, loss, opt, batch_size):
        train_ds = CTDataset(data, labels)
        train_dl = DataLoader(train_ds, batch_size=batch_size, shuffle=True)
        epoch_loss = 0

        ...

        for epoch in range(NUM_EPOCH):
            print("epoch %d / %d" % (epoch + 1, NUM_EPOCH))
            for i, (x, y) in enumerate(train_dl):
                opt.zero_grad()
                pred = model(x)
                loss_value = loss(pred, y)
                loss_value.backward()
                opt.step()
                epoch_loss += loss_value.item()

            if epoch_loss <= 1e-5:
                break

            print(f"loss/epoch: {epoch_loss}")
            epoch_loss = 0

    ...
```

First, we pass the model to the function that we want to train, followed by the labels, which represent the actions of the rule-based agent, and the data, which corresponds

to the states. Then, we provide the loss function and optimizer, and finally, the batch size. We found that a batch size of 1024 worked well for us, especially as the model had more data to train on when the ReplayBuffer became fuller. We did experiment with other batch sizes, but if the batch size was too small, it took a long time for the loss function to converge to 0 (which it didn't entirely reach).

We achieved good results when the `num_epoch` was initially set at 2000, which we later increased to 5000 during training. Additionally, we sometimes had to remove older data from the ReplayBuffer to ensure that the model continuously learned from new games. It was also worthwhile to reduce the view box to 7x7 because it significantly reduced the number of possible states. After 4-5 days of training, our model became reasonably proficient at playing the game. While it couldn't perfectly mimic the rule-based agent, as the agent occasionally had to make random moves when there were no more items in its 7x7 view box, we still found the results quite satisfactory.

Afterward, we attempted to enhance the model using Q-learning. For this, we utilized the reward function from our best agent in the first project. Unfortunately, this approach didn't work well; the agent's performance deteriorated, and eventually, it struggled to play the game effectively. We saw improvement when we alternated training the model with the rule-based agent and then applied deep Q-learning see Figure 3. However, even in this scenario, the results were significantly worse compared to when our model was solely imitating the rule-based agent.

In the end, we were unable to improve the agent using Q-learning beyond the level of our initial project, which we naturally submitted. We also considered reasons why we couldn't enhance the agent pre-trained with rule-based data using deep Q-learning. The main issue likely stems from optimizing the model with vastly different values when it imitates the rule-based agent compared to when it undergoes deep Q-learning. In the first case, the model is optimized with one-hot encoded actions of the rule-based agent, while in the latter case, it's optimized with the values of the reward function and the next states.

5 Experiments and Results

5.1 Deep Q-Learning Agent

In this section we are giving an overview of the experiments we have done with our deep q-learning model that lead to our best performing agent. The basic framework is already described in the sections Methods and Training (give reference here). Hence at this point we trained the model, evaluated the performance and based upon that we changed hyperparameters or auxiliary rewards to improve the model. We always conducted a similar training routine, namely starting off with a small field with height and width 7 containing coins on every tile, so 21 coins in total. This gave us the opportunity to already evaluate if there are flaws in the implementation or how fast the network converges with the chosen hyperparameters. Then we scaled the field up to 11×11 and increased the difficulty by including crates. Finally the agent was trained with all settings as in the target field, so a 17×17 field, 9 coins and a crate density of 0.75. The last step was to add opponents to train in the exact same environment our agent will be tested in. Simultaneously to raising size and difficulties we also increased the steps per round from 60 to 200, as the agent does not need 200 steps to collect all coins in a field as small as 7×7. (EVTL. SOLLTE DAS IN TRAININGS PART)

Initially most experiments resulted in the fine tuning of the rewards and auxiliary reward functions. A good indication that the rewards were not helping the model to get trained in the most efficient way was a big deviation between the sum of the rewards in comparison to the score achieved by the agent. Ideally the score in the end of a round is constantly quite high if the sum of the rewards is high, especially if the sum is in a high range after each round for many rounds already, so if it is converged. The plots in Figure 4 show an example of this phenomenon when the model was trained for 10000 rounds in the 7×7 coin-heaven scenario with coins on every tile . There is a quite sudden improvement in the score between the 2000th and 3000th round of training and this jump can also be observed in the rewards plot on the left but is much less significant. Furthermore the rewards plot still increases and seems to have converged after the 5000th round of training. Nonetheless the score is getting worse and keeps fluctuating between the highest possible score 21 and a score as low as 3. When letting the agent play we noticed that this happened because the agent dropped many unnecessary bombs and thus killed itself occasionally, so the agent was not able to collect all the coins. However the total amount of rewards was still high because the agent was rewarded a lot for outrunning a bomb and was not punished for dropping a totally unnecessary bomb. To understand this kind of behavior the logged messages were really helpful. Therefore we fine tuned the rewards with a closer examination of matching the extent of rewards for different actions with each other. We also added a penalty for dropping bombs that will not destroy crates or are in reach of an opponent.

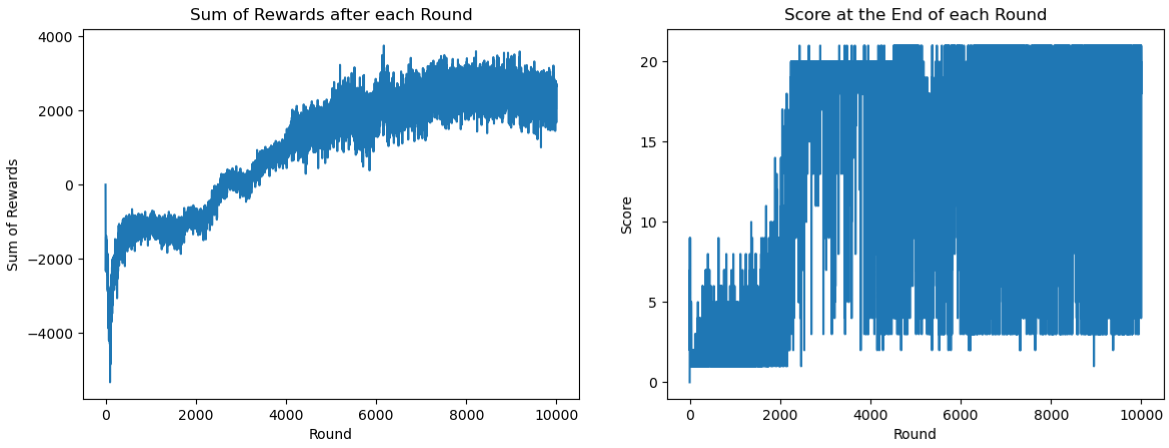


Figure 4: Both plots were generated during the training of the Deep Q-Model on a small coin-heaven scenario with coins on every free tile. The x-axis of both plots represent the rounds of the training. While the left plot depicts the sum of all rewards given to the agent in each round, the right plot shows the score after each round. These plots show discrepancy between rewards and scores.

After a little bit more trial and error and some more improvements of the rewards we were already able to produce a model that collected all coins efficiently. Figure 5 shows the corresponding rewards and scores of 8000 rounds of training. This time score and reward coincide very well. Again between the 2000th and 3000th round the agent makes a jump and apparently found a good way to handle this environment. Henceforward, both scores and rewards, are on a constantly high level. Of course there is a certain amount of fluctuation as the agent is supposed to do some random moves during training to find a potentially better way of playing. Here the probability of

doing a random move during training, not calculated by the network, was constantly 10%.

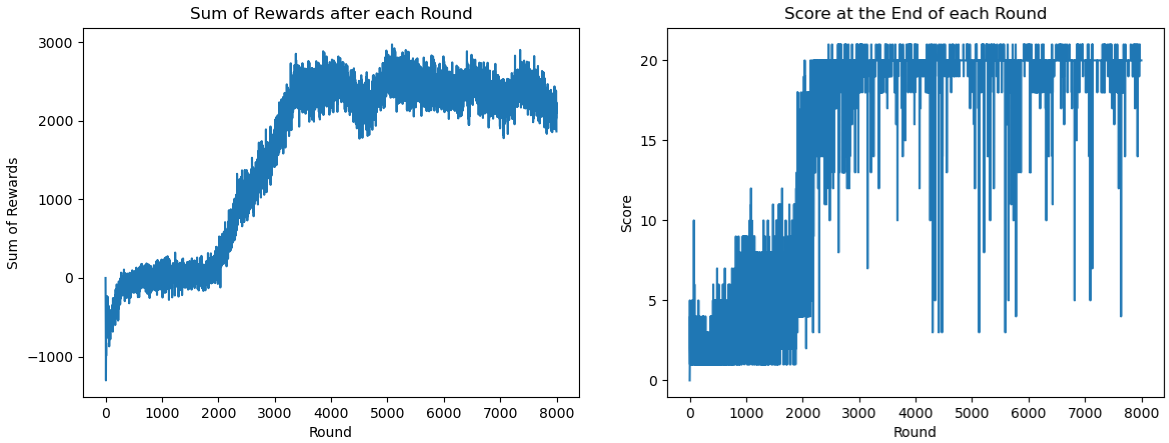


Figure 5: The plots are constructed in the same way as before, but after reward fine-tuning the scores shown here fit to the high reward sum.

Although the auxiliary reward functions seemed to be elaborated very well at this point the agent did not play smoothly on larger fields and in the harder levels even after a lot of training rounds. The main concern was that the agent often kept killing itself in early steps of the game. With the assumption that the rewards are given reasonably we also had to evaluate other hyperparameters. Our neural network consisted of a very simple architecture so we thought about features to add to the network to enable better learning. Even though dropout is a very common thing to have in neural networks, it did not make sense for our task as it is used to prevent overfitting, which cannot happen here. Instead we included batch normalization which should stabilize and accelerate the training. Unfortunately when doing the training from scratch the batch normalization did not result in a better running agent or a faster training, probably because the batch size was too small. Not only was the batch size small in general with 256, but we could not collect enough training samples if the agent killed itself early in the game. This did not allow the positive effects of batch normalization to unfold in our network.

Next we gave different loss functions a try. We initially used the smooth L1 loss but also thought of trying the cross-entropy loss because it is the standard loss for classification models, where the output usually is a list of probabilities. In our case the output gives the probability of the action to be the best action to take in the current state of the game for every possible action. Applying the cross-entropy loss showed that our model is not suited for this kind of loss. A reason for this could be that we want to find a good metric to calculate the loss between the predicted and the target q value, calculated following the Bellman equation. The literature hints that we should minimize the quadratic error between those values. In the formula for the cross-entropy loss the quadratic error is never calculated. Instead losses that can be considered next to the Smooth L1 loss is the closely related Huber loss and of course the MSE (mean squared error) loss. In the end MSE loss proofed to be most efficient, so our final agent was also trained with this loss. When discussing the loss we also inevitably gave thought to the optimizer. The AdamW optimizer worked well but RMSprop was more effective. Especially the faster convergences shown in Figure 6 was a decisive point in

favor for the RMSprop optimizer. Thus we proceeded with RMSprop but as AdamW achieved good results too we believe that it would haven been able to run similarly when giving more consideration to the learning rate.

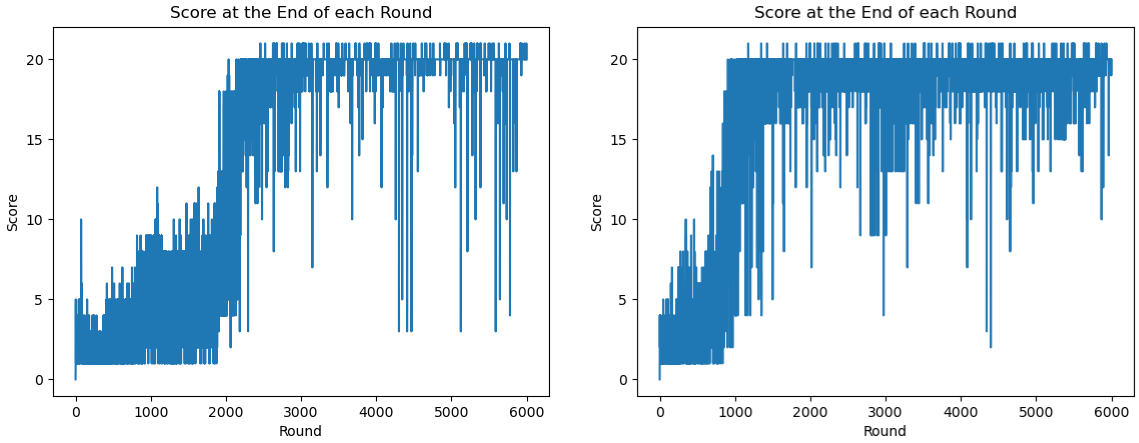


Figure 6: The plots are constructed in the same way as before, but both plots depict the scores of to different trainings. The left one was trained with the optimizer AdamW and the right one was trained with RMSprop. The model using RMSprop improves and converges faster.

With the configurations chosen as explained above we reached a fairly good playing agent after 6000 rounds of training in coin-heaven (7×7 field, coins on every free tile and 60 steps per round), 60000 rounds of training in loot-crate (11×11 field, 20 coins, 0.4 crate density, 120 steps per round), 15000 rounds of training in classic (17×17 field, 9 coins, 0.75 crate density, 200 steps per round) and 40000 rounds of training in the classic scenario as described before but with two rule based agents and one coin collector agent as opponent. Strangely, except of being able to play the game well the agent did not learn to not do invalid actions. We observed that the majority of self killing happened because the agent still did invalid action, like trying to move in a direction where a wall is located to escape a bomb. Thus to get a better playing agent we forced the agent to not do invalid actions in the not-training mode. After each model output we check if the chose action, so the action with the highest probability, is possible and if not we take the action with the second highest probability and check this one and so on.

Finally, we created an agent that can compete with a rule based and coin collector agent but is mostly not able to beat them. Still we can conclude that our model did learn how to play the game as it clearly outperforms a random agent, which nearly always makes 0 points. Figure 7 shows the scores of four agents, namely our reinforcement learning agent, a rule based agent, a coin collector agent and a random agent, when they play against each other in the classic scenario for 10 rounds. We also found that our agent performs better against stronger agents, maybe because it was trained against strong agents. When we chose the same opponents as during training, so 2 rule based agents and 1 coin collector, our agent often outperforms at least one of them.

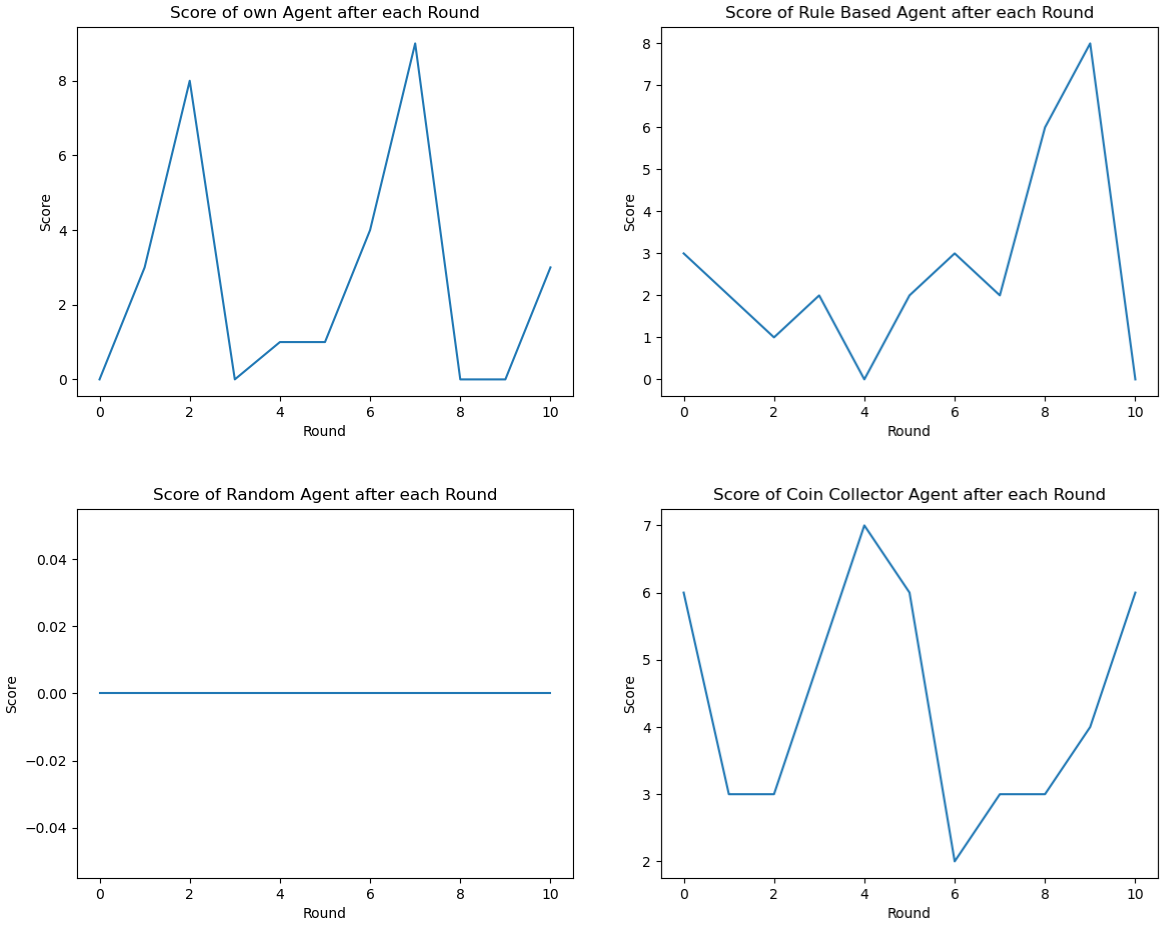


Figure 7: These plots result from one game of bomberman with 10 rounds. Our reinforcement learning agent played against a rule based agent, a coin collector agent and a random agent. While the random agent did not score any points, our agent and the rule based agent both scored 29 points in total and the coin collector won with 46 points.

notes:

- 2 Aufteilungsmöglichkeiten, entweder wie die unterschiedlichen optimierungen vom trainingslevel her waren, also aufteilen in coins, crates und opponent training und verbesserungen oder für jede Trainingsänderung
- of course in training should be some randomness but the low score came from the agent killing itself which was due to dropping many unnecessary bombs because outrunning the bomb was more effect in regards to the rewards than only collecting the coins
- in coin heaven scenario with 7x7 field left a nearly perfect result, where the agent collects all coins very efficiently, also shown in plots (Git3, destroyer of universes, neuer reward 5, 1. Training)
- in these plots we can also see a clear point where the agent, randomly picked a very good path and kept improving on that till final converging, except some outliers due to randomly chosen action in training
- at this point the probability for randomly chosen action was constant at 10%
- took logs as help to understand what happend during training, and what the agent learned to do because of the rewards
- although at this point the rewards seemed pretty good there were troubles when going into the bigger training scale with crates and opponents and bigger fields

- one of our main concerns was the fact that the agent killed himself very often, and sometimes quite early (plots rewards 5, 12. Training, clarify that this was not only training, but the last 10000 rounds of training), again very flactuating, basically only 0 score so often because killing self before getting the chance to collect many coins
- when assuming that the rewards are given in a way that enables the agent to learn the game efficiently, we had to evaluate other hyperparameters and check if they enable better and more learning
- a very basic nn architecture, common features to add in nn is e.g. dropout but as this is to prevent overfitting, which is not a problem in reinforcement learning we didn't apply that
- batch norm unfortunately didn't do anything for us at the point we tested it, which probably was caused by too low batch sizes in general but especially because the batch size also depended on the amount of rounds saved in our memory class, and if the agent killed in itself early the batch probably was as small as 3 samples, question is would batch norm give us a big advantage anyway, as it is more helpful in deep networks to stabilize and make the training faster
- we added batch norm before addition of the multiple models and before activation as suggested by
- loss initially started with smooth l1 closely related to huber, also tested huber which converges to mse loss which worked very well (the final agent was trained with mse loss but a very similar result would probably also be possible with huber/smoothl1), cross entropy was not working very well, literature says squared loss of target q and predicted q value
- optimizer adamw was effective, rmsprop in combination with mse loss showed the best results and also faster convergence, maybe if fitting learning rate better adamw would be able to come up with same results, convenient not to bother with the learning rate (plots Git 3 mse rms 1. Training vs reward 5 1. Training, schnellere konvergenz in scores, ähnlich für rewards)
- excluded invalid actions because logs showed that this happens when trying to escape bomb, agent should be able to learn this (cannot go into walls) but we definitely want to avoid it even if it is not learned (create plots of agent that avoids invalid action vs not avoids invalid actions)
- batch norm, loss (huber, cross entropy, mse, smmothl1), optimizer changed (AdamW vs RMSProp), excluding invalid action in play modus (not during training), more memory, batch size (256) and randomness threshold calculated dependent on round (unfortunately detected error very late, that it was dependent of step in round instead of round, so did it manually, starting with hihger probability like 0.1 and going down to 0.05 and 0.025)
-plots final one running against random and rule based (maybe already do that earlier, as proof that we are on right track)

6 Conclusion

References

[1] COMP THREE INC. *Variational Autoencoders are Beautiful*. https://www.compthree.com/blog/autoencoder/. 2019.