Reinforced Regrets: A Report on the Tribulations of Creating a Bomberman Agent with Reinforcement Learning

Ole Plechinger

Heidelberg University, Germany
ole.plechinger@stud.uni-heidelberg.de

Thomas Wolf

Heidelberg University, Germany
thomas.wolf01@stud.uni-heidelberg.de

Contents

I. Introduction

In this report, we describe the approaches, challenges, and solutions we encountered during the final project of the Machine Learning Essentials course, creating an agent that can play the game Bomberman against agents from other teams.

Bomberman (described in more detail in Section 2.1) is an ideal task for studying and experimenting with Reinforcement Learning due to its turn-based nature and simple game state. It creates many tricky situations and difficult decisions that allow good models to differentiate themselves.

Our GitHub repository[1] uses only common Python libraries including PyTorch, Numpy, and matplotlib. The final agent can be found in the *Crow_of_Reinforcement* folder. The report is written in the LaTeX alternative Typst and can be viewed at [2].

II. Background

A. The game

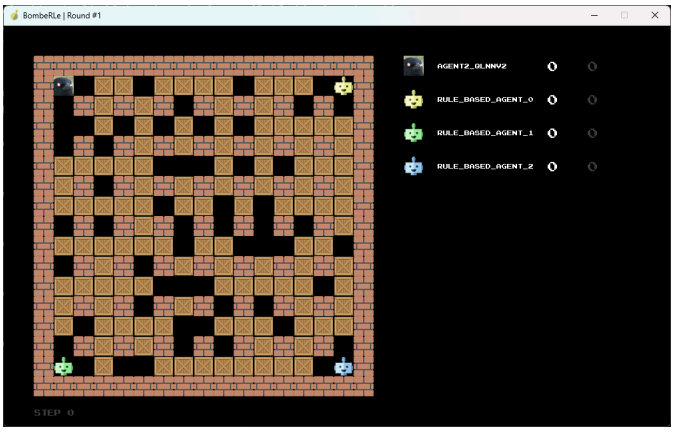


Fig. 1: Initial Position of a Bomberman Game

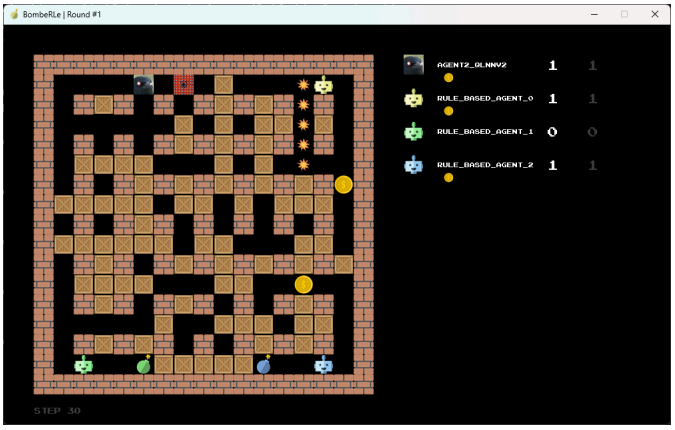


Fig. 2: Midgame of a Bomberman Game

The game Bomberman is a simple video game where four independent players (referred to as agents) navigate across a 17x17 grid with the goal of getting the highest average score among its opponents.

The game is turn-based with each agent choosing one of six possible actions every move: Moving up, down, left, right, Doing nothing, or Placing a bomb. The score is primarily gained by collecting coins, which are found randomly within crates that can be blown up with a well-placed bomb. Bombs explode three tiles in every direction unless blocked by a wall, a few turns after they have been placed. Other than destroying crates to reveal the coins they may hide, bombs can also be used to kill agents, including the one that placed the bomb in the first place. So agents need to consider how to get to cover in time and when it is possible to trap an opponent.

B. *Why not just brute-force the best action?*

Unlike games like "tic-tac-toe" which has only 362,880 [3] possible games; Bomberman has, with four agents picking from six actions for up to 400 turns, an upper limit of $\left(6^4\right)^{400} = 1,1 \times 10^{1245}$ possible games. We can also estimate the number of game states: $(15 \times 15) - (7 \times 7) = 176$ tiles are not blocked by a wall (see Fig. 1) and may host a crate, minus four times the three corner tiles that never host crates, thus even ignoring agent positions, bombs, and coins there are $2^{176-4*3} = 2.3 \times 10^{49}$ possible states. This of course includes many nonsensical game progressions and some invalid game states, but it should be clear that brute-forcing the decision tree is computationally infeasible.

So how can this problem be solved anyway? The key lies in the property that the relationship between the game state and the corresponding optimal decision is not random. By looking at a large representative set of games, which actions were taken, and whether those actions led to a win or not, the optimal decision can be approximated; this is what the machine learning approach is all about!

The following sections cover two general machine-learning approaches we tried out.

C. *Random Forests*

To understand a forest, one must first understand the trees that compose it. Each tree is a binary decision tree, and each node of the tree along the decision path receives a random subset of features used to make the decision. This approach suffers from some drawbacks[4]. Instead of just one tree, Random Forest[5] uses an ensemble of trees to make decisions: $y = \frac{1}{N}\sum_{i=1}^{N} T_i(x)$, where N is the number of trees in the forest, T_i are individual trees, x are the features and y is the prediction. This increases predictive accuracy, lowers variance, and improves generalization compared to a single decision tree. While usually used for supervised learning and classification tasks, we tried to transfer this to our use case with limited success, see Section 3.4.2.

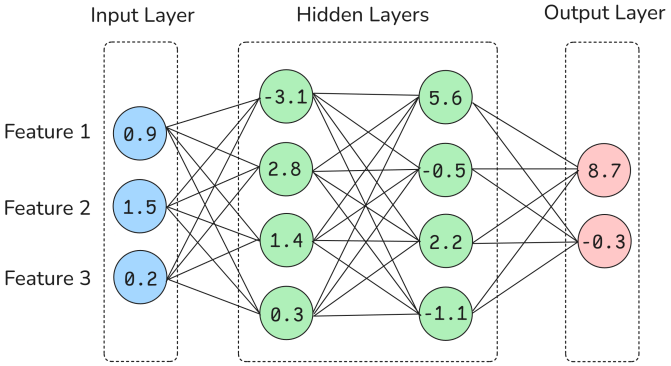
D. *Neural Networks*

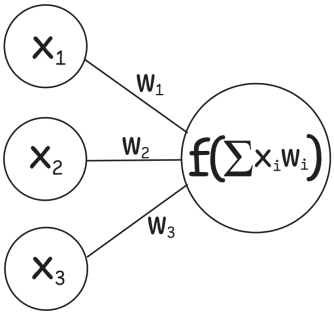
A Neural Network [6] is a machine learning approach inspired by biological brains, that is very good at learning complex relationships but generally requires more training data to learn from. Neural Networks consist of layers of nodes called *neurons*, each holding a value.

The first layer of neurons, the *input neurons*, get their value directly from the input, in our case, this can be a numerical representation of the entire game state or some *features* of the game state like distance and direction to coins, whether a bomb can be placed or whether the agent is currently in the vicinity of a bomb.

For every next layer the neurons, called *hidden neurons*, get their values through a *weighted sum* of the previous layers' neurons followed by a non-linear *activation function*.

And the values of the last layers' neurons, the *output neurons*, are a representation of the output we want to learn.





By adjusting the weights in the weighted sum any complicated function can be approximated, so the weights are also called the *parameters* of the neural network, and a full set of parameters is called a *model*. The activation function is necessary to represent non-linear functions because without it a neural network would just be a series of linear transformations that result in another linear transformation.

To figure out what good parameters are we need some representative data to learn from. If we know what the output to some input should be we can compare it to the models' output and calculate a *loss*, for example, the *Mean Squared Error*: $\frac{1}{n}\sum_{i=1}^{n}(\text{output}_i - \text{correct-output}_i)^2$, which is the value we want to minimize in the average over all training data. We can improve the loss by differentiating the loss with respect to every parameter and adjusting the parameters according to this gradient. This method is known as *gradient descent* and can be expressed with the following equation where β is a parameter and γ the *learning rate*:

$$\beta_{\text{new}} = \beta_{\text{old}} - \gamma \frac{\partial\ \text{Loss}}{\partial \beta} \mid \beta_{\text{old}} \tag{1}$$

E. Markov Decision Process

In order to use Random Forests or Neural Networks to learn Bomberman we need a formal framework in which to describe it. This framework is a Markov Decision Process:

The setup consists of an Agent and an Environment. The Environment is at any time t in some state S_t and given an action A_t there is a fixed probability for a next state S_{t+1} and reward R_{t+1} to happen. The Agent should learn to choose actions that maximize the reward.

F. Q-Learning

Throughout a game, we will then collect the state-action pairs as well as the corresponding rewards and next states and use that data to train a model to predict the reward that each action would provide in the current state.

This alone is not quite enough to learn to play well, because most actions do not immediately result in a reward, but only much later. We can fix this by predicting the total expected *return* of an action throughout the rest of the game instead of just the reward of one time step.

In the following equation $\hat{Q}$ is the expected return of an action A_t in the state S_t, which consists of the reward R_{t+1} that was awarded for A_t and a discounted estimate of the game after:

$$\hat{Q}(S_t, A_t) = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) \tag{2}$$

This will cause rewards given late into the game to propagate to the earlier actions, enabling the learning of actions that pay off in the long term.

G. Convolutional Networks

If you give the model the full 17x17 arena, with one input neuron for every position, the spatial structure of the arena is not reflected in the input. An event happening at one position would not make it clear that it could happen at other positions as well and would be the same event. Also spatial patterns like what happens in the position *next to* an agent's could not be learned.

To fix this we can use *Convolutional Neural Networks*. A modification of regular Neural Networks where we, instead of doing a weighted sum over every neuron of the previous layer, apply a 2D convolution. A convolution takes a 2D array of numbers and outputs another 2D array of numbers with potentially different dimensions, by applying a *kernel* to the input. We will not go into detail here, but a convolution essentially acts like a weighted sum of the surrounding pixels. This enables the recognition of spatial patterns and because the same kernel is applied at every position this also generalizes learned events over any position.

III. Project planning

A. Resources and libraries

Training a Neural Network can be computationally demanding, however, because we never trained huge models, with the largest one being 10 layers deep, and because we both have decent graphics cards (Nvidia RTX 3060ti/3080) we did not have to offload the computation to a cloud service.

Across our training runs and experiments, the training speed varied depending on the model size, how many agents were taking part, and how long the agents were able to survive from about 1 $\frac{s}{\text{it}}$ (Iterations per second) to 10 $\frac{s}{\text{it}}$, one iteration being one entire game round.

We used the popular Python IDE PyTorch and the libraries PyTorch, NumPy, and Matplotlib because we were both already familiar with them from the exercises and personal experience.

B. Teamwork

As detailed in the next section, because of exams and an internship our work times did not overlap too much. Thus, we primarily worked on separate features of our agents and regularly discussed our progress and plans during evening Discord meetings.

For the report, Thomas wrote an outline of the things we intended to mention and we assigned the different headings to ourselves, based on who knew most about the topic or just preferences, by marking each heading: for Ole, for Thomas. Some headings where we both wrote significant parts are assigned both avatars.

We used GitHub to synchronize and manage the project's code and a Typst document to keep track of todos and write the report.

C. Time planning

We began our work by creating the GitHub repository on August 11, 2024. The submission deadline was six weeks and two days later.

I had exams on Aug 23 and had to prioritize studying, so the initial project setup was done by Thomas, afterwards, I was able to spend my time pretty freely on the project. On Sep 30 (the same day as the report submission) I had another exam, combined with the difficulties in finishing the agent and writing the report, the last couple weeks of the project were extremely stressful.

Time was scarce, (40h/Week Internship, exams at the beginning) so I just went for the trusty "If time, work on the project and hurry" strategy, intending to have a competitive agent ready about two weeks before the deadline to be safe. I sacrificed my social life, my sleep schedule, and the breaks necessary for my mental health for this project.

D. Progress timeline overview

a) *Familiarization with the framework and code structure:*

To implement our agent we were provided with a framework: every agent is in a separate folder in a folder `agent_code` and requires the files `callbacks.py` and `train.py`, which need (or recommend) the following functions:

- `setup()`: called once to set up the agent and other variables
- `setup_training()`: the same as `setup()` but for training runs
- `act()`: called every time step and needs to return the action the agent chooses
- `game_events_occurred()`: called every game step, so that the data can be stored and used for learning
- `end_of_round()`: called at the end of a game, to do stuff like saving the model
- `reward_from_events()`: a function to assign events a reward or punishment (negative reward) that is given to the model every time the event occurs
- `state_to_features()`: a function that transforms the given game state into the features that are passed to the model

The framework allows you to run different agents against each other, optionally in training mode and in different scenarios that consist of different arenas. The most important scenarios are:

- *Coin Heaven*: A map full of coins without crates so the agent can learn to collect coins efficiently without having to deal with bombs
- *Loot Crate*: A map with many crates that contain coins more often than usual to learn to blow up crates without dying
- *Classic*: The default scenario that the agents will later compete in

b) *First try: bad random forest agent:*

For our first simple implementation of the random forest agent, the features were not as advanced. Specifically, nothing was stopping the agent from killing itself. Accordingly, it usually immediately killed itself, then learned not to drop bombs, and started to collect coins in coin-heaven mode simply by moving straight forward and getting stuck in a corner. Interestingly, when one is not careful with the training rewards, killing oneself can be the best action for the agent to maximize rewards/avoid losses. We realized that the random forest approach would only work for very simple features, and would not be able to compete with neural networks when it comes to complex feature spaces. Thereafter, this approach was abandoned and work began on the heart, or rather brain, of the project:

c) *Q learning neural network first implementation:*

Based on this [7] tutorial and this paper [8], we built a basic Q-learning agent class, together with its brain, the neural network class, and adapted initialization and training rewards accordingly. Henceforth, these three files shall appear in every agent folder:

1) `callbacks.py`, for initialization and the `act` method that is called for getting a decision from the agent based on the game state
2) `train.py` , handing out training rewards based on the game state, the chosen action, and its consequences
3) `agent.py`, which contains the code that makes up the actual agent: the model architecture itself and the `state_to_features` function and all the functions called by it, responsible for deriving the features from the game state

Similar to the random forest agent, initially, the Q-learning agent's performance was abysmal, since it mostly killed itself quickly, and then learned to avoid placing bombs and merely managed to collect a few coins at the border of the map coin-heaven mode. We knew that this agent would never stand a chance when it comes to blowing up crates without killing itself, not to mention fighting enemy agents, and thus conceived two solution proposals:

1) Generate training data by letting rule-based agents play against each other and use that to train our agent.
2) Forbid the agent to die by designing a function that disallows actions that would lead to death.

We chose the second solution and implemented such a function that deterministically helps the agent escape its own bombs (in single-player mode). This function greatly accelerated our progress and would be developed much further later in the project. It shall henceforth be known as the function of immortality (see Section 5.3.2). It allows us to fully train an agent in merely 100-150 iterations/games, enabling a rapid iterative design process.

d) *Learning to collect all the coins* :

To tell the model in which direction it would have to move to get to the closest coin and to support many other input features we developed a few utility functions which would later be called `get_distance_map()`, `get_nearest_objects()` and `get_direction()`. See Section 5.2.1 for more information. This enabled the agent to get perfect scores on the coin-heaven scenario.

e) *Loot-crate* :

Thanks to the function of immortality, after learning to collect coins this was quite straightforward. We simply added two features: the number of crates a bomb dropped on the current position would reach, and the direction to the nearest free tile next to a crate detailed in Section 5.2.4 and it worked straight away.

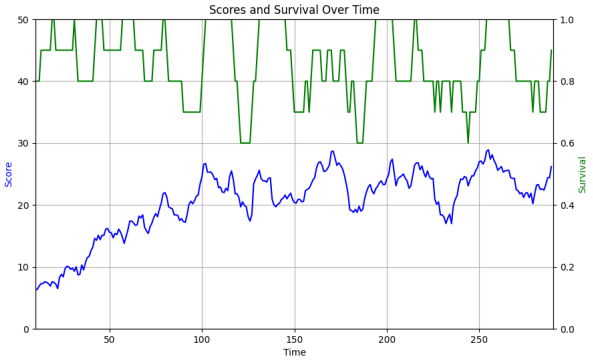


Fig. 3: Example plotter output

f) *Plotter* :

To keep track of the agent's progress we made a simple utility `plotter.py` that plots the values from a 'scores.csv' file (standard `matplotlib`). Initially, it would only create a static plot after training, later we made it auto-update to watch the progress during training, and later, we added the survival rate.

g) *All-Plotter* :

There are even more useful metrics: Kills, suicides, and especially the cumulative training reward the agent achieves during a game. All those metrics are plotted using the `all_plotter.py`. It also allows the simultaneous use of varying smoothing window sizes, allowing a more precise illustration of the agent's progress. Nevertheless, the original plotter did not become superfluous, since it is more apt for a quick check on the agent's progress during training due to its simplicity.

h) *Convolutional Agent and structural Improvements* :

Using only non-convolutional features means that the agent can seldom access the full information of the game board. Therefore we wanted to add support for convolutional features in addition to regular ones (we are calling them *linear features*)

To do this effectively several improvements to the code structure had to be made: Instead of passing individual layer dimensions as separate arguments to the model initialization, which makes changing the layer dimensions require changes to several different parts of the code, the model's input and output dimensions are passed as a single dictionary and the model initialization and inference is generalized to work with different architectures. Now the model passes convolutional features through convolutional layers and then combines the last convolutional layer with the linear features which are then passed through linear layers to the output layer. The dimensions of each layer and other architectural choices are now exclusively made in the model initialization function.

Also because the training data is getting too large with convolutional features we had to split the memory from the model to be able to store them in separate files.

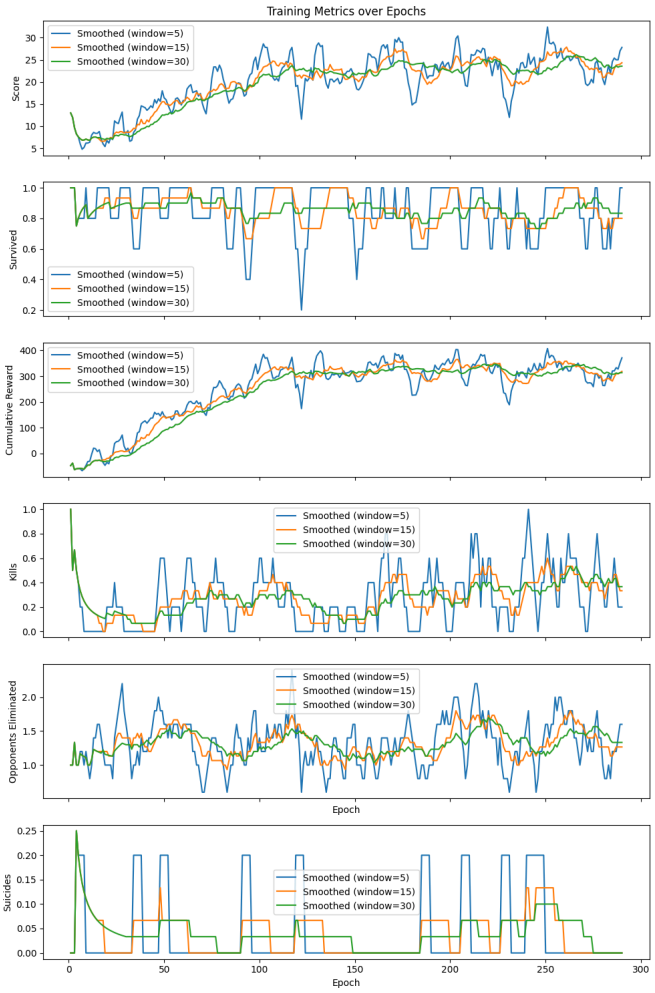


Fig. 4: Example all-plotter output

With these modifications to the framework, we can now pass the distance map and direction map from the above sections as convolutional features as well as other useful maps like a coin map, crate map, agent map, and danger map.

A feature we ended up not finishing was going to be a `focus_map` *function, which zooms every map to a smaller radius around the agent. This would make the model smaller and thus easier to train and also make it easier for the model to get information about relative positions.*

i) *Data Augmentation* :

In Bomberman, the orientation of the game is irrelevant so the agent should make the same choices in symmetric scenarios. This is done in `state_symmetries()` by applying transformations like rotating or flipping to the convolutional features and directional linear features when retrieving a state batch from memory.

j) *Refining the features and function of immortality* :

Back to the non-convolutional simple Q-learning agent: After adding a few features to help the agent deal with enemies (detailed in Section 5.3) and refining the function of immortality to consider enemy agents, our agent quickly became quite competent at dealing with rivals. Our agent can now finally defeat the rule-based agent.

TABLE I: PARAMETERS USED FOR OUR FINAL AGENT

Parameter	Designation in project code	Value
input dimensions (features)	input_dims	60
hidden layers dimensions	l1_dims - l5_dims	512, 512, 256, 128, 128
output dimensions (features)		6 (since there are six possible actions)
dropout rate	dropout_rate	0.1
initial exploration rate	epsilon	1.0
epsilon decay	eps_dec	0.01
final exploration rate	eps_end	0.03
Learning Rate	lr	0.001
gamma	gamma	0.9
batch size	batch_size	64
maximum memory size	max_mem_size	100000

k) *Bugfixes, further refinement, training, comparison against other agents* :

At this latest stage of development, the focus was on maximizing the agent's performance for the competition. Thus, many bugs were fixed, especially in the function of immortality. During this time, another feature was born that turned out to be quite useful: the `direction_suggestion` function (see Section 5.3.1) decides on the current goal tile of the agent, who is rewarded for following the direction.

We started fine-tuning the training process, and for this purpose created a `train-script.py` file in the root folder, which enables automatic training in various scenarios, which would require multiple shell commands to execute, without the need for a human to be present to manually start the training process for each chapter after the previous one finished, which saved quite some time and helped us stay organized.

l) *Final agent* :

We trained lots of different agents with different rewards and training routines and decided on the one that performed best. See Section 4.3 and Section 6.1 for further details.

IV. METHODS

A. Model architecture Q-learning agent

The following will describe the most well-known and frequently used type of neural network, a *multi-layered perceptron*, and explicate the parameters in Table I.

a) *Network:*

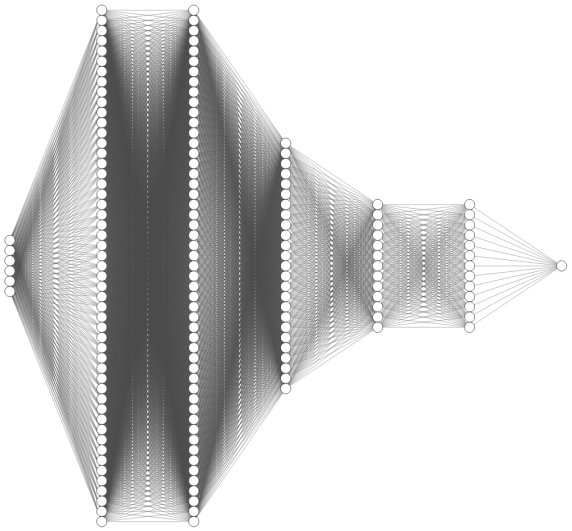


Fig. 5: Visualization of our chosen architecture, with one-tenth of the neurons per layer displayed (hence only one output neuron and six input neurons)

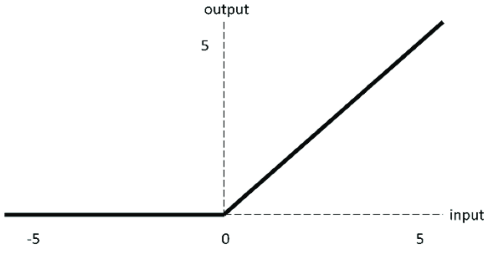


Fig. 6: ReLU(x) = max(x, 0)

The network itself was made using `torch.nn`. In Table I the network layer sizes are listed: the input layer has 60 neurons, since the feature vector has a length of 60, and the output layer consists of six output neurons, because there are always six possible actions. Why did we choose five hidden layers and those specific layer sizes? In Section 6.2.1 we find that significantly smaller networks work fine too. The idea behind the larger network was that it would be able to generalize better and learn more complex correlations between the features. The layers decrease in size with depth like a funnel, with the intention that the early layers extract a wide variety of lower-level features that arise from the interplay of the sixty input features (see Table II), while later layers become progressively smaller because they are responsible for learning higher-level abstractions and decision-making based on the features extracted by earlier layers. This follows the intuition that moving deeper into the network, learned representations are refined and combined into more abstract and meaningful features, which requires fewer neurons. Additionally, having large layers in the later stages of the network can lead to *overfitting* because the network would have too much capacity to memorize the training data without generalizing well to unseen states. Thanks to [9] Fig. 5 shows what our network would look like at $\frac{1}{10}$th the scale.

b) *Activation function:*

We used the standard ReLU (Rectified Linear Unit) activation functions from `torch.nn.functional` as displayed in Fig. 6. ReLU is simple, reliable, and resistant to the *exploding/ vanishing gradient problem* [10]. We checked whether the problem known as "*dying ReLU*", where neurons that learn large negative weights never activate again and thus are effectively dead, occurred in our model, by testing *leaky ReLU* functions, but could not discern a difference.

c) *Dropout:*

Dropout is a regularization technique where, during each training iteration, a random subset of neurons is "dropped out" (i.e., set to zero) to prevent over-reliance on specific neurons and encourage more robust feature learning. This helps reduce *overfitting* by ensuring the network generalizes better to unseen data, as it cannot rely on any specific subset of neurons to make predictions. High dropout rates hindered our model's learning progress, and too small dropout rates have too little effect, and thus we decided on a moderate value of 0.1, meaning that in each iteration ten percent of the neurons are "dropped out".

d) *Exploration vs exploitation:*

Exploration refers to the agent trying new, potentially suboptimal actions to discover more about the environment, learn new strategies, and find better long-term rewards.

Exploitation is when the agent chooses actions that it already knows to yield the highest immediate reward, based on its learned knowledge.

Too much exploration slows convergence, while too much exploitation risks missing out on discovering better strategies.

We use a simple ε-greedy policy during training, starting with the initial exploration rate $\varepsilon = 1.0$ and reducing it linearly every iteration with an epsilon decay value of 0.01 until it reaches the final exploration rate $\varepsilon = 0.03$. Starting with $\varepsilon = 1.0$ ensures maximum exploration at the beginning of training, meaning the agent takes completely random actions. This is important early on because the agent has no prior knowledge of the environment, so exploring widely helps it learn diverse strategies and avoids getting stuck in suboptimal behaviors. We arrived at the value of 0.01 for epsilon decay via empiric testing, it is a simple tradeoff between getting enough experience for good generalization versus completing training fast. As a result, it reaches the final exploration rate at iteration 97, hence the score value in most of our training score diagrams increases until that point since the rate of random actions decreases and the agent can act out its learned policy.

e) *Experience replay:*

Experience replay is a technique where the agent stores its experiences (transitions between states, actions, rewards, and next states) in a memory buffer during training. These experiences are then sampled randomly in batches to update the model. The main benefit of experience replay is that it breaks the tempo-

ral correlation between consecutive training samples, helping the agent learn more efficiently by training on a diverse set of experiences. This reduces the risk of overfitting to recent experiences and helps the model generalize better. We considered implementing *prioritized experience replay*[11], but our implementation worked fine without that and we did not find the time to try it.

The *maximum memory size* defines how many experiences (transitions) the agent can store in its memory before it starts discarding old experiences. This would theoretically allow for $\frac{100000}{400} = 250$ stored games, more than enough for us to finish training without discarding experiences.

Batch size refers to the number of experiences randomly sampled from the memory during each learning step to update the neural network's weights. We decided on a batch size of 64 as a tradeoff between training speed and the ability to generalize well.

f) *Learning:*

As explained in Section 2.6, *gamma* (γ) is the discount factor in reinforcement learning, and determines the importance of future rewards by discounting them as they become more distant in time. We empirically decided on $\gamma = 0.9$, meaning a reward received in the next time step is worth 90% of an immediate reward.

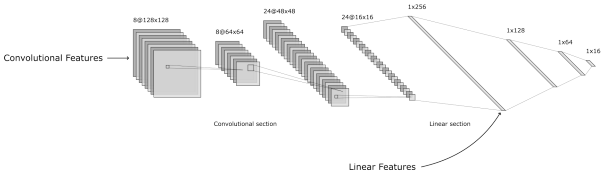
The *learning rate* controls how much the model's weights are adjusted during training when computing the gradients in backpropagation. We empirically chose a learning rate of 0.001 together with an exponential learning rate scheduler from `torch.optim.lr_scheduler` that multiplies the learning rate by a factor of 0.99 in each step. This helps the agent explore faster early in training when it knows little and then refines its learning with smaller steps later to converge toward an optimal policy. Without scheduling, a fixed high learning rate could lead to overshooting, while a fixed low learning rate might make learning too slow or stop before reaching a good solution.

B. Model architecture convolutional agent

The agent is separated into three main classes:

- `DeepQNetwork`: containing the model weights and dealing with the model architecture and inference
- `StateMemory`: containing and managing the training data
- `Agent`: combining the two classes above and doing higher level functions including choosing actions and learning

The model can take convolutional and linear features handled as a dictionary of two lists. The convolutional features are put through convolutional layers and pooling layers first and then put through linear layers together with the linear features.



C. Performance metrics

Up until the later stages of the project, simply looking at the plotter output was enough for our purposes. But toward the end, a more accurate technique was needed. Therefore, when we wanted to compare the performance of agents, we ran 100 games (against each other or against rule-based agents) with the `--save-stats` argument and compared the metrics in the .json files created in the results folder.

V. Training

A. Data Augmentation

For the convolutional agent, we apply all symmetries of a square to convolutional features and directional linear features.

B. Helper functions that are not directly features

a) *distance map:*

`get_distance_map()` uses the Dijkstra pathfinding algorithm to calculate the distance and direction from the agent's position to any other position. The distances are a 17x17 array of values, with 999 being reserved for inaccessible positions like walls. The directions are a 17x17 array of representations of one of the four directions the agent would have to move in to follow the shortest path to that position, or no direction if that position is inaccessible or the position of the agent itself.

The distance and direction map are used for many other utility functions and features.

b) *get nearest object directions:*

`get_nearest_objects()` uses the previously calculated distance map to sort a list of positions by distance and returns the top choices.

Together with the direction map, this is used to get the direction to the nearest objects of some type.

c) *get safest direction:*

We faced the problem, that after laying a bomb, our agent often chose a bad direction to escape, ran into enemies or bombs, and died as a result. If the agent is not currently on a bomb, the `get_safest_direction` function returns a zero-vector of length 4. Otherwise, a cone-shaped area of tiles for each direction that is not blocked by walls or crates is formed, and the number of tiles of that cone containing enemies, bombs,

or ongoing explosions is calculated, where bombs and enemies have a higher danger score than explosions. The cone with the smallest cumulative danger is in the safest direction, which is returned as a one-hot.

d) *get direction to nearest create* :

A slight modification of the `get_nearest_object_directions` function, returns the direction to a free tile *next* to a crate, instead of the crate itself.

e) *coins collected* :

The `coins_collected` function keeps track of how many coins have been collected so far in the round and returns their number, which is also saved in the agent itself. It works by storing the scores of each agent and comparing the current scores to the previous scores in every step.

f) *danger map* :

Returns a 17 by 17 map of the field with danger levels: 0 for no danger, up to 4 for active explosions / immediate death. For each tile in a bomb radius, the danger level is calculated as follows: danger level = 4 - bomb countdown. This means it starts at 1 when the bomb is dropped and goes up to 4 when the bomb explodes in the next step.

g) *get safe tile distances* :

The function `get_safe_tile_distances` returns a vector of length four, one entry for each direction. Every entry is an integer signifying the distance to the nearest safe tile in that direction, considering bombs and even whether enemies could block that tile. If no safe tile is found that is close enough for the agent to escape there before being blown up, and the function is not called for the purpose of predicting whether there would be a reachable safe tile if the agent dropped a bomb on the current position, the function does a second pass, allowing tiles that could be reached by enemies quicker or have a lower danger level, meaning they are still in a bomb radius, but of another bomb that would explode later.

h) *dead end map and list* :

The function `get_dead_end_map` iterates over every tile in the field, if the tile has one open neighbor, this means exactly one of the four neighboring tiles is not blocked by a wall or crate, it might be the closed end of a dead-end. The dead end is then traced, it ends when a tile does not have exactly two open neighbors. The dead-end map is initialized as zeros in the shape of the field, and the tiles that are part of dead ends are marked with ones. In the dead-end list, there is an entry for each dead-end containing the closed end, open end, tile before the open end, the entire path (all tiles that belong to the dead end), and whether there is an enemy in the dead end.

C. Features

TABLE II: FEATURES

Feature	Number of values	Explanation
suggested direction	4	Section 5.3.1
blocked actions	6	The result of the function of immortality, Section 5.3.2
agent position	2	$0.1 *$ (agent x, agent y)
is at crossing	1	Section 5.3.3
is repeating actions	1	-
direction and distance to nearest coin	4	Section 5.3.4
direction and distance to second-nearest coin	4	Section 5.3.4
neighboring explosions/ coins	5	-
bomb available	1	Whether it is allowed to drop a bomb
number of crates in bomb radius	1	-
is in danger	1	Is in radius of a bomb
enemies distances and directions	12	Section 5.3.5
enemies relative positions	12	-
is next to enemy	1	-
direction to enemy in dead end	4	Section 5.3.6
can bomb enemy in dead end	1	Section 5.3.6
$\sum$	60	

In this section, we will briefly explain a selection of our features, focussing on the most impactful and significant. Many features are passed as one-hot encodings because this representation allows categorical data (e.g., directions or actions) to be processed in a format suitable for neural networks, ensuring that each category is treated as distinct and equidistant, without implying any ordinal relationship between the categories. This helps the model learn more effectively by preventing unintended hierarchical associations between feature values.

a) *suggested direction* : , The function `direction_suggestion` decides on a "goal" tile and returns the direction to it as a one-hot of length four (since there are four directions). During training, the agent is rewarded if it moves toward the goal and is punished if it does not. To decide on the goal tile the function proceeds as follows:

1) Is there a trapped enemy nearby? If yes, return direction to it.
2) Is there a reachable coin on the field? If yes, return direction to it.

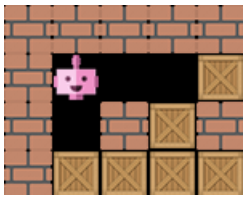
3) If it is on a bomb, return the safest direction to escape using Section 5.2.3
4) If there are coins left in crates (see Section 5.2.5) return the direction to the nearby tile (search radius 4) where most crates would be reachable by a bomb dropped there, calculated in the `get_directions_to_most_reachable_crates` function, which is not explained in this report as it is fairly straightforward. If there is no such tile nearby, return the direction to the nearest crate using Section 5.2.4.
5) If there are enemies left, return the direction to the nearest enemy. This could be improved for the tournament by storing enemy names and how "killable" they are across games, and then returning the direction to the enemy that is the easiest to kill, see Section 7.2.
6) If the function reaches this point, only our agent and perhaps a few empty crates are left on the field. If crates exist, the direction to the nearest crate is returned.

This function significantly improved our agent's performance by motivating it to blow up remaining crates and attack enemies even towards the end of the game, which our agent previously struggled with.

b) *function of immortality* :

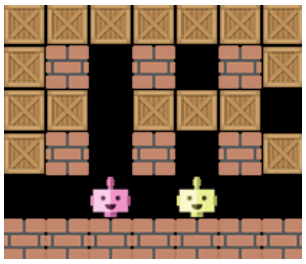
This is the most useful function in our project and had a huge impact on our progress (see Section 3.4) as it enables much quicker training. It always returns a vector of length six, one entry for each action: [right, down, left, up, wait, bomb] If the entry is "0", the action is safe/allowed. If it is "1", somewhere in the function of immortality it was decided that this action could doom the agent to die. Not only are these features, but are also directly used to block the disallowed actions in the `choose_action` function of the agent. These are the steps the function goes through to arrive at the final vector of disallowed actions:

1) For each direction, check whether the agent would immediately die due to an explosion if it went there or whether that direction is blocked by a wall/crate/bomb.
2) If it is in danger, in the explosion radius of a live bomb, the distances to the nearest safe tiles for each direction are calculated, see Section 5.2.7. If the nearest safe tile is too far away for the agent to reach before the bomb blows, this direction is disallowed. If waiting or dropping a bomb would make the agent miss the window to reach the nearest safe tile, those entries are also set to "1" in disallowed.
3) If the agent can drop a bomb, check whether it would be able to escape, by calculating what the danger map from Section 5.2.6 would look like if the bomb was dropped and calling the safe tile distances function from Section 5.2.7. If there would be no safe tile reachable if it dropped a bomb, disallow the bomb action. Example:



If the agent dropped a bomb in this situation, there would be no reachable safe tile.

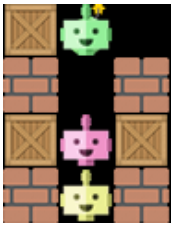
4) Do not enter a dead end if an enemy could trap the agent, or leave the current dead end immediately if an enemy would otherwise be able to reach the exit of the dead end before our agent. This is determined using Section 5.3.6. Example:



If the pink agent entered the dead end above it, and the yellow agent went left, the yellow agent would be one step away from the tile before the dead end, just like the pink agent. If in the step after that, the yellow agent gets to move onto that tile, it could drop a bomb there in the next step that the pink agent could not escape.

5) If the `is_repeating_positions` function returns true and the "bomb" action is allowed, disallow all other actions and thus force the agent to drop a bomb, as explained in Section 6.6.2

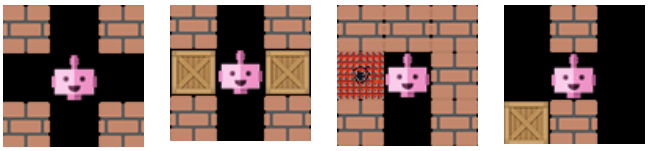
Our agent cannot be killed, if there is only one enemy alive. If there are more, it can get pinched by two others:



Shortly before the project deadline, we thought of a solution for this, which works by considering passages whose farther end could be blocked by an enemy as dead ends, but it was buggy and thus left out. Even with that, our agent would not be truly immortal, as there are still complex multi-agent situations where it can be slain.

It should be noted that we tried to disallow as few actions as possible in this function, while still letting it fulfill its purpose since we do not want to accidentally prohibit valid and good

TABLE III: CROSSING?



actions. We do, however, make the assumption that dying is a fundamentally bad strategic move.

We recognize that the use of this function may be morally ambiguous, as the lecture is about machine learning and not engineering a rule-based agent. However, the quest for immortality was fun, frustrating, fulfilling, challenging, captivating, and totally worth it. We argue that our agent still clearly has a significant learning curve (see Fig. 7), even learning to survive better, it can make its actions freely and is only influenced when absolutely necessary for survival, learning its own way of playing beyond that. This is still within the spirit of this project.

See Section 6.4 for an analysis of how dependent our agent is on the function of immortality and a discussion about the disadvantages and potential negative impacts on performance.

c) *is at crossing :*

This feature returns "True" if the agent's x as well as y positions are uneven numbers. This is the case for the first three examples in Table III, but not for the last. But what is its purpose? The agent is punished for dropping bombs at tiles that are not crossings since the bomb will have less impact there because two sides are blocked off by walls. Thus, this feature helps the agent place bombs more effectively.

d) *direction and distance to (second) nearest coin :*

Use the distance and direction maps from Section 5.2.1 to get information about the two nearest coins. This is encoded as a one-hot scaled by the square root of the distance.

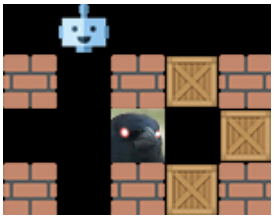
e) *enemies distances and directions :*

For each enemy, a one-hot vector of length 4 is calculated, which is non-zero only in the direction to said enemy (calculated using Section 5.2.1) scaled by the square root of the distance of that enemy but capped at a maximum value of $\sqrt{30}$. Those one-hot vectors are concatenated to one vector of length 12. If there are less than three enemies, it is padded with zeros.

f) *dead end features :*

The function `get_direction_to_enemy_in_dead_end` uses the dead-end list from Section 5.2.8. For every dead end in the dead end list, if there is an enemy in it, this function checks whether our agent could plausibly reach the tile before the open end before the enemy does. If this is the case, the first four entries of the vector of length five that this function returns will be a one-hot showing the direction to the enemy in the dead end. If the enemy is in a straight dead end and our agent is three or fewer steps away from the closed end, one of these three or fewer tiles containing the enemy, the fifth element of the returned vector is set to "1", else "0", this is the "can bomb enemy in dead-end" feature. This means, that if our agent dropped a bomb now, the enemy would not be able to escape (unless another bomb destroys a crate of the dead end, enabling the enemy agent to flee). This is used to prevent our agent from following the enemy too far into a dead end, giving the enemy a chance to drop a bomb first and thus kill itself, for which our agent gets no points.

The function "`do_not_the_dead_end`" works similarly, but instead checks whether it is our agent that is in danger of being trapped in a dead end. That is the case if the distance to the tile before the dead end of our agent plus one is larger or equal to the distance of the closest enemy to that tile. If so, our agent is forced to escape the dead end in the function of immortality. Example:



The distance of our agent to the tile before the dead end is one, since it is inside the dead end, on the open-end tile. The distance of the blue agent to that tile is two. Thus, our agent must move left, because if it waits or moves right and the blue agent moves down, the blue agent could move on the tile before the open end in the step after that and trap our agent.

D. *Auxiliary rewards and the reasoning behind them*

In this section, we explain the rewards and punishments from Table IV we used to reinforce positive behaviors that increase the agent's chance of success, e.g. make our agent learn a desirable policy. Note that for some rewards, there is a punishment for doing the opposite. This is to prevent the agent from being able to accumulate rewards by moving back and forth, see Section 6.6.2.

a) *Movement punishments:*

Small penalties for moving in any direction. The goal is to prevent random or excessive movement and encourage more strategic movement patterns. The small penalty incentivizes the agent to seek meaningful actions (e.g., collecting coins or attacking enemies) rather than wandering aimlessly. A larger penalty for waiting, since this action is often passive and can indicate indecision or inaction. It discourages waiting unless absolutely necessary (e.g., avoiding explosions).

b) *Why no punishment for suicide:*

When our agent kills itself, it still receives the "got killed" punishment. We do not want to punish suicide more than getting killed by an enemy, since it is strategically better to commit suicide than to let an enemy have the kill, because the enemy

TABLE IV: AUXILIARY REWARDS

Event	Reward	Explanation
survived round	10	
got killed	−10	
killed self	0	Section 5.4.2
invalid action	−1.5	Agent chose an action that could not be executed, usually trying to move onto the same tile as an enemy
moved up, down, left or right	−0.1	Section 5.4.1
moved toward enemy in dead end	0.7	Section 5.3.6
ignored enemy in dead end	−0.3	Section 5.3.6
waited	−0.3	Section 5.4.1
waited on a bomb	−1	
waited in explosion zone	−0.3	
is near border	−0.1	It is easier to die near the border of the map, especially the corners
is repeating actions	−0.5	Section 6.6.2
did opposite of last action	−0.2	Section 6.6.2
followed direction suggestion	0.3	Section 5.3.1
did not follow direction suggestion	−0.2	Section 5.3.1
coin collected	8	
bomb dropped	−0.75	Section 5.4.3
dropped bomb that can destroy crate	1.5	Section 5.4.3
dropped bomb that can destroy crate bonus for amount	0.5	Section 5.4.3
dropped bomb not at crossing	−0.5	Section 5.3.3
dropped bomb while enemy near	1	Section 5.4.3
dropped bomb next to enemy	1.5	Section 5.4.3
is next to enemy	−0.3	A neighboring tile contains an enemy
killed opponent	10	
opponent eliminated	4	Also reward making enemies kill themselves
enemy got kill	−5	Avoid learning to help enemies get kills for the opponent eliminated reward
balance reward	0.1	Section 5.4.4

would get five points for the kill. Our agent is indeed punished more for getting killed than for killing itself due to the "enemy got kill" punishment.

c) *Bombing rewards:*

There is a variety of possible rewards when a bomb is dropped. First of all, there is a baseline punishment for dropping bombs, so there is a negative reward when the bomb is not useful, teaching the agent to use its bombs effectively.

We want to encourage the agent to clear crates efficiently and thus find many coins, to do that we first have a basic reward for dropping a bomb that will destroy a crate, and then give a bonus reward of 0.5 for each crate the bomb can potentially blow up. This works well together with the "number of crates in bomb radius" feature which tells the agent how many crates would be reachable by a bomb dropped at its current location.

To reward attacking enemies, we have the "dropped bomb next to enemy" and "dropped bomb while enemy near" rewards. The latter comes into effect if an enemy was within three tiles of our agent when the bomb was dropped, and is rewarded twice if the enemy was within two tiles.

d) *Balance reward:*

This is applied every time rewards are calculated, like an event that happens every step. If the overall rewards are too high, the agent has too little incentive to end the game quickly and becomes 'lazy' and inefficient. If the overall rewards are too low/negative, it will have little incentive to avoid dying.

TABLE V: FINAL AGENT TRAINING SEQUENCE

Step	Scenario	Number of rounds	Agents in play
1	loot-crate	50	2 Crow of Reinforcement, 2 rule-based agents
2	loot-crate	50	Crow of Reinforcement, rule-based agent, q learning tree agent[12], coin collector agent
3	classic	50	Crow of Reinforcement, rule-based agent, q learning tree agent[12], coin collector agent
4	classic	10	Crow of Reinforcement, rule-based agent, q learning tree agent[12], user agent (human)

Fig. 7: Final agent score plot during training

E. *Training methods*

This is the four-step training sequence our final agent went through we decided on after some testing: Table V

The dramatic drop after step 100 in Fig. 7 is because, in the loot-crate scenario, there are fifty coins available, while in the classic scenario, there are only nine. The ten matches with a human are not in the training scores diagram, since the purpose of this was for the human to bring the agent into new and unusual situations to enable it to generalize better against a wider range of opponents, and thus the scores for these rounds are skewed. Each training iteration took somewhere between three and nine seconds, depending on the agents in play and the rate of eliminations.

VI. Experiments and Results

A. *Performance comparisons*

In this section, we try to assess how good the performance of our agent is by comparing it to the rule-based agent and human performance. We do not compare it to the other weaker predefined agents since it is much better than the rule-based agent. For brevity, we will only display the bar charts for relevant metrics here, although we do have a diagram maker python file that creates such bar charts for all metrics in the .json files that are created by executing several rounds with the –save-gui argument.

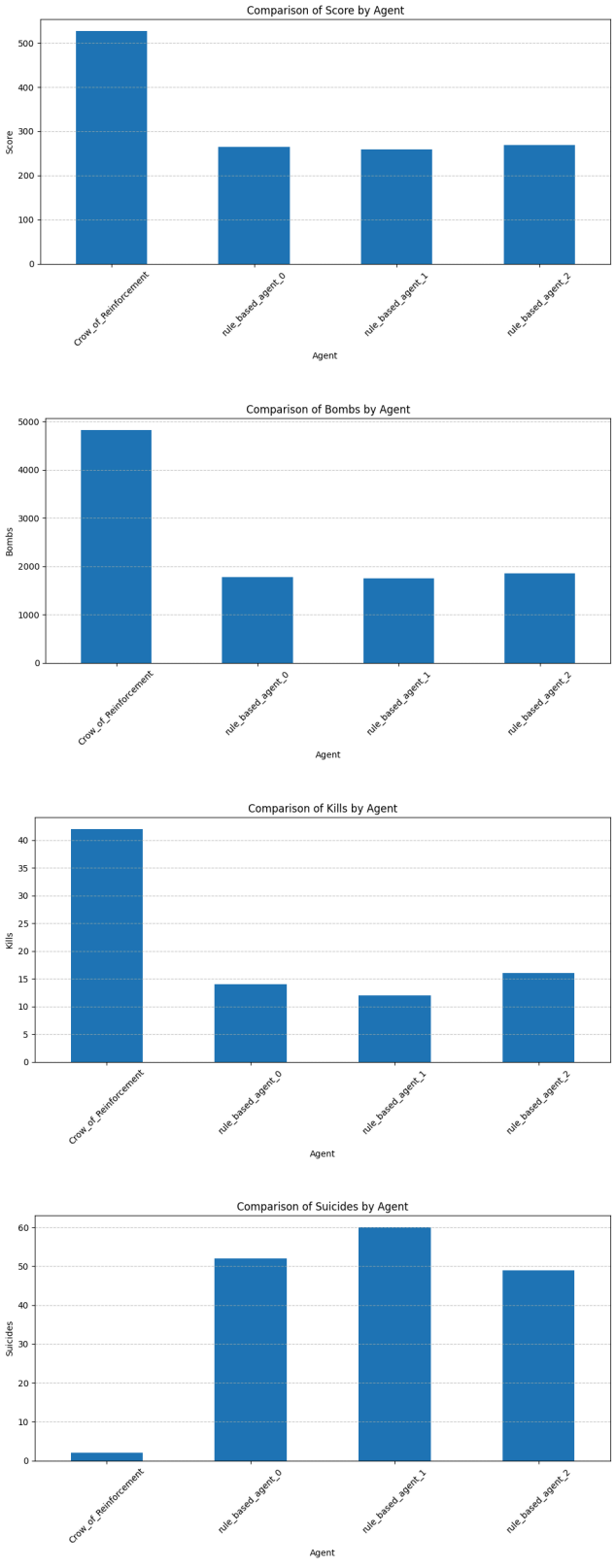
Comparison to rule-based agent

From the comparison with the rule-based agents in Table VI, we can tell that our agent is significantly better, scoring almost twice as high as the rule-based agents. Our agent frequently drops bombs and generally avoids self-destruction, although it did commit suicide in two instances. Our agent gets to kill a rule-based agent almost every second round and presumably makes them kill themselves just as often too, likely due to the dead-end features in Section 5.3.6 exploiting the biggest vulnerability of the rule-based agent. However, this comparison is not apt to predict how well our agent will perform in the tournament since it learned to fight rule-based agents during training and might not perform as well against unseen agents.

Comparison to human expert

As can be seen in Fig. 8, a skilled human player can outperform our agent. Without “human errors” like not thinking the action through, accidentally clicking the wrong button, etc. the score difference would be significantly larger. Humans have a much better overview of the entire field, can predict enemy movements, learn their behavior, and exploit their weaknesses. For example, rule-based agents tend to run into dead ends, a human can predict when a rule-based agent is about to run into a dead end and move in that direction early, while our agent simply does not have features that give it that information. A human expert can also find more efficient ways to blow up many crates, strategically leave dead ends for enemies to enter, and other advanced strategies. Our agent performs all right given its limited features, but much better agents are possible, see Section 7.2.

Table VI: Final agent vs. three rule-based agents, classic, 100 rounds



B. *Small experiments*

a) *Different layer sizes and number of layers* :

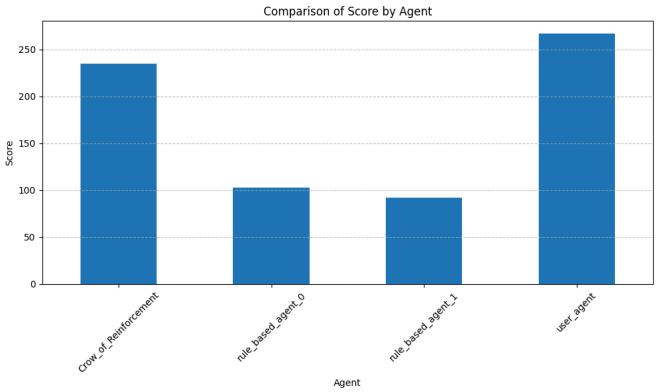


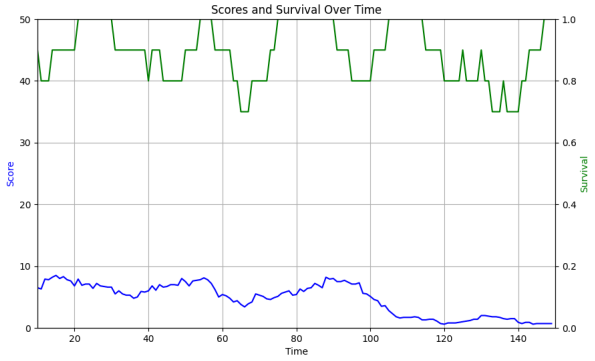
Fig. 8: Performance comparison human expert vs final agent vs two rule-based agents, classic, 50 rounds

We tried several different model dimensions to see how this would affect the performance. Each run was 150 rounds of the default scenario with two ML agents and two rule-based agents, the base case is 5 layers with 128/128/64/64/6 dimensions (with the last layer being the output layer). However neither doubling nor halving the amount of layers noticeably affected performance. The large size of the layers was also not necessary, with 32/32/6 performing just as well. Only at model sizes of 16/16/6 and smaller the performance dropped by a large amount.

b) *Experiment: sparser rewards :*

In this experiment, we only reward collecting coins (+1) and killing enemies (+5) balanced with a punishment of −0.01 per step and apply the same training sequence as for our final agent, see Table V.

This is the resulting train score plot:



Our agent does not converge if the rewards are too sparse because it receives insufficient feedback to learn which actions are beneficial or detrimental. Without frequent and timely rewards, the agent struggles to establish clear cause-effect relationships between its actions and the outcomes, making it difficult to optimize its behavior or improve its policy over time.

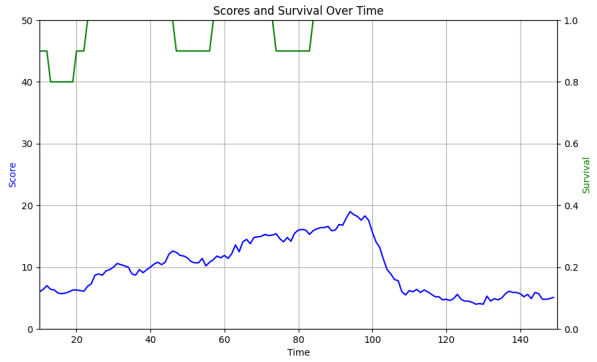
C. *An investigation of network shape: is decreasing layer size with depth advantageous?*

To determine the effects of decreasing layer sizes, we will execute the same training sequence with the same training rewards we used for our final agent in Table V, but with the following layer sizes:

1) 512, 512, 512, 512, 512
2) 128, 128, 256, 512, 512 (inverted funnel)

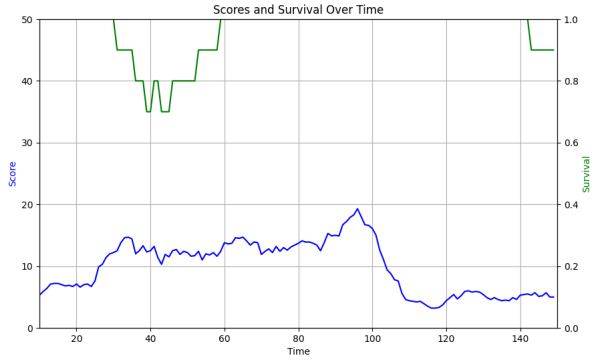
We hypothesize that since we assume that layer sizes decreasing with depth like a funnel better the performance of the model, 1) should perform worse than our final agent and 2) even worse than 1).

Result 1)



In 100 classic games against three rule-based agents, it got a cumulative score of 476, which is indeed worse than our final agent with a score of 527.

Result 2)



In 100 classic games against three rule-based agents, it got a cumulative score of 479, unexpectedly not worse than the agent with equal layer sizes. However, a more scrutinous look at the results revealed that this agent collected a lot fewer coins, but got seven more kills, which might just be a coincidence. To determine that, we let those two agents play 100 rounds against each other, with two rule-based agents: Agent 1) scored 372, and agent 2) scored 407. So, it was likely not a coincidence. But why would the inverted funnel shape be better? Perhaps the

difference is little and it just learned a better policy by coincidence. If we had more time we could explore this in more detail.

Conclusion: Although the difference is small, our final agent with the decreasing layer sizes does seem to perform better than the configurations tested here. An explanation proposal can be found in Section 4.1.

D. *Is the function of immortality necessary?*

When deactivating the function of immortality detailed in Section 5.3.2 before training, the agent is unable to learn, if we deactivate it after training, it will almost immediately die by suicide and perform very badly even after longer training. Perhaps our training rewards are biased toward an agent that can survive reliably and are not suitable for an agent that first has to learn not to kill itself. For now, our agent heavily depends on the function of immortality.

It is also possible that the function of immortality hinders the development of better features: It may cause the agent to have less pressure to predict potential danger because it can just rely on the function of immortality to do that job.

A solution proposal would be gradually reducing the influence of the function of immortality and giving negative rewards for trying to do something that is blocked by the function of immortality. The latter did not work on its own, and we did not get to try the former.

E. *Performance on GPU vs CPU*

Interestingly, we discovered during Docker tests towards the end of the project that inference is faster on the CPU than on the GPU. This may be because the time to do the GPU calculations could be so short that it is not worth it to move the data to the graphics card and back. This is supported by the fact that training is faster on the GPU, presumably because the additional calculations of back-propagation change the bottleneck.

F. *Difficulties and how we overcame them*

a) *The agent kills itself too often and does not learn:*
Solution: function of immortality, see Section 5.3.2.

b) *Oscillation problem or being stuck:*
This is a recurring issue we faced during the project: the agent often learns to move back and forth in one place or follows other repetitive patterns, signifying that the agent is stuck. We implemented several remedies:

- Ensuring that rewards and penalties are balanced and the agent has nothing to gain by oscillating
- A training punishment for doing the opposite of the previous action.
- A punishment for following a pattern that signifies that the agent is stuck, whether that is the case is determined by looking at the last four actions in the `is_repeating_actions` function.
- Saving the last 20 locations of the agent in a deque, and if there are too few unique positions (< 5), meaning the agent is likely stuck (see `is_repeating_positions` function), force it to drop a bomb. It will escape the bomb due to the function of immortality and move away from the location where it became stuck.

The latter approach proved the most effective. However, we encountered the oscillation problem again at a very late stage of development, despite the previous remedies. Turns out, a feature that passes the agent's previous action as a one-hot (which was supposed to help prevent oscillations) is the culprit of this problem. After leaving out that feature, the oscillation problem was solved. We are clueless as to why using the agent's previous action as a feature would cause the oscillation problem, especially since it was heavily punished for oscillating and the cumulative training reward per game went down as epsilon (see Section 4.1.4) decreased and fewer actions were taken randomly, meaning that even random movement should be a more desirable policy for the agent than oscillating.

c) *Confusing, complex, unordered, and buggy code:*
There was no real solution here, we simply cleaned, commented, and ordered the code and fixed the bugs. A lot of the chaos could perhaps been prevented by disallowing oneself to stay awake until 4 am trying to finish that one functionality with complete disregard toward code cleanness.

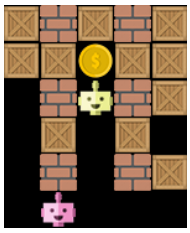
VII. Conclusion

A. *Summary*

This report highlights two main models: the Q-learning neural agent and a convolutional neural network (CNN) agent. The Q-learning agent, named *Crow of Reinforcement*, performed well after significant tuning, feature engineering, and additional helper functions like the function of immortality to prevent the agent from killing itself. The CNN agent was not fully implemented due to time constraints. The Q-learning agent showed competent performance. It outperformed the baseline rule-based agents but was still weaker than a skilled human player as it lacked the features for some advanced strategies.

B. *Possible improvements for our agent given more time*

- Our agent still dies sometimes, but achieving true immortality without limiting the agent's actions too much should be possible.
- Support for more advanced strategies, like not collecting coins inside a dead end to bait enemies to enter it:



- More features (e.g. bomb positions), trying to lessen the advantage that convolutional agents have over our non-convolutional implementation.
- Making the convolutional agent work, it has a lot more potential to achieve near-optimal, superhuman performance. The best possible agent must be capable of observing the entire game state.
- Given that the agent can learn to play well after only 100 iterations, collecting 100 great games played by a meat-based mortal as training data and then training the agent on that data could result in a better agent.
- Somehow adapt to the opponents during the tournament, remembering their weaknesses across games and exploiting them. Remembering specific opponents should be possible since our agent can get their names from the game state.
- There are many strategies we haven't even had the chance to look into due to temporal constraints, e.g. gradient boosting, prioritization of experience replay, genetic algorithms, NEAT, LSTM cells, …

C. *Improvement suggestions for game setup for next year*

- different rule-based agents: it would be nice to have multiple rule-based agents of different strength to compare ML agents to or to use as opponents for the ML agents to learn from.
- Allowing bigger uploads to MaMpf or offering a different upload mechanism: Some models can get quite large and the file size limit is only about 20-30mb. Unless this is an intended limitation it would be nice to have the option to upload larger files.
- It would be nice to be able to run a tournament locally with our implementation using all the agents of the competition from a previous semester, for example, to test whether our agent can adapt to agents during the tournament to exploit their weaknesses if that functionality had been implemented.

D. *Final Words*

We wish the other teams good luck in the tournament! May the best agent win.

References

[1] "Our GitHub repository." [Online]. Available: https://github.com/togewolf/BombermanML

[2] "Our report on Typst." [Online]. Available: https://typst.app/project/wkHA9pShFFkYnMovbJ7xC-

[3] "Wikipedia: Game Complexity." [Online]. Available: https://en.wikipedia.org/wiki/Game_complexity

[4] "Scikit-learn: Decision Trees." [Online]. Available: https://scikit-learn.org/stable/modules/tree.html

[5] L. Breiman, "Random Forests," 2001, [Online]. Available: https://link.springer.com/article/10.1023/a:1010933404324

[6] "Wikipedia: Neural Network." [Online]. Available: https://en.wikipedia.org/wiki/Neural_network_(machine_learning)

[7] "Torch Q-Learning Tutorial." [Online]. Available: https://www.youtube.com/watch?v=wc-FxNENg9U

[8] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," 2015, [Online]. Available: https://www.nature.com/articles/nature14236

[9] "Neural network visualizer." [Online]. Available: http://alexlenail.me/NN-SVG/index.html

[10] K. Sharma, "Vanishing/exploding gradients problem," 2023, [Online]. Available: https://medium.com/@kushansharma1/vanishing-exploding-gradients-problem-1901bb2db2b2

[11] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized Experience Replay," 2015, [Online]. Available: https://arxiv.org/abs/1511.05952

[12] "GitHub repository of the ql_tree agent of another team from last year we used for training purposes." [Online]. Available: https://github.com/victories12/bomberman_hu_xu/blob/main/agent_code/ql_tree_agent/