Deep Reinforcement Learning Group Project

January 22, 2025

Group:

- Marvin Kohnen
- Moritz Gehring
- Julius Ferber

Code and results can be found at `https://github.com/MarvinKohnen/deep-reinforcement-learning/tree/main/drl_deep_project`.

1 Exercise A

1.1 Basic Training Routine: DQN

We first implemented a DQN agent, given the full state space on the classic tournament map. Furthermore, we adjusted the existing q_learning file with our own values. After mutliple iterations, the best performing model ran with the following values and rewards:

Model stats:

- ϵ start at 1
- ϵ end at 0.05
- ϵ exponential epsilon decay of 100.000
- γ is 0.99
- 50.000 episodes

Rewards:

- COIN_COLLECTED: 1
- KILLED_SELF: -5
- INVALID_ACTION: -0.5

- KILLED_OPPONENT: 5
- CRATE_DESTROYED: 0.1
- GOT_KILLED: -5
- WAITED: -0.1

The model showed signs of learning (as can be seen in Figure 1 rewards (top) and episode length (bottom)). When running the model, this learning manifests in the agent moving in corners to increase survival rate, while neglecting coin collection, crate destruction or aggression towards other players. Moreover, the model is quite unstable, often self-killing the agent rather early on during the run.

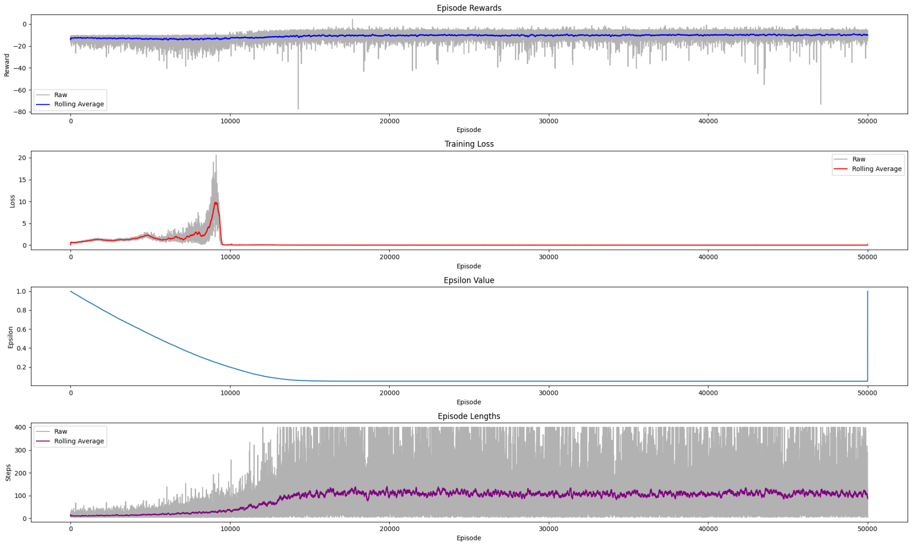

Figure 1: Results of simple DQN Training, plots for reward, training loss, epsilon decay, and episode length over 50000 Episodes

The Code for this model is found in the folder "simple_dqn".

2 Exercise B

2.1 Reward Shaping

Our first approach to improve the DQN was to use different rewards, as this is the easiest way to influence the model behaviour. We realised, that the event KILLED_SELF and GOT_KILLED would punish our agent twice. Therefore we dropped the former from our reward structure. Furthermore, we increased the reward for coin collection and also introduced a reward for coin found and crate destroyed. Ultimately, these were the specific values that resulted in the best outcome:

Rewards:

- COIN_COLLECTED: 2
- INVALID_ACTION: -0.5
- KILLED_OPPONENT: 5
- CRATE_DESTROYED: 1
- COIN_FOUND: 1
- GOT_KILLED: -5
- WAITED: -0.025
- BOMB_DROPPED: -0.0125
- MOVED_UP: -0.0125
- MOVED_RIGHT: -0.0125
- MOVED_DOWN: -0.0125
- MOVED_LEFT: -0.0125

This reward shaping also introduces a time penalty for every action, punishing the agent for any action it takes. The time penalty for moving is $-5/400$, i.e. the reward for getting killed divided by the maximum episode length.

The idea behind this is that the agent should still prioritise surviving, as the cumulative reward for surviving the entire episode is still greater or equal to the reward for dying, i.e. the agent should never end in a local minimum of -5 by just blowing itself up immediately. The negative reward for waiting is doubled, encouraging the agent to do aything of value (allowing for more exploration of the state space) unless it leads to a negative reward (i.e. walking into an explosion). The idea behind the time penalty in general is to introduce an intrinsic motivation for the agent to seek a positive reward, which in this case can be achieved simply by destroying crates or discovering coins.

2.2 Curriculum Learning and Catastrophic Forgetting

We implemented various approaches to curriculum learning in our project. As stated earlier, our model was initially trained on the full tournament map, but we subsequently experimented with reduced-scale environments (e.g., 7x7 or 9x9 grids) to simplify the learning process. However, the models trained on these smaller maps failed to scale to larger environments, even when the scenario remained unchanged, rendering this approach ineffective.

Our next strategy involved developing less complex scenarios within a 17x17 grid, which is equivalent in size to the tournament map. These simplified scenarios proved to be more approachable, allowing for faster training and yielding promising outcomes (as illustrated in Figure 2, shwoing the curriculum coin scenario).

Nevertheless, we encountered a significant challenge: when attempting to transfer the weights learned from these simpler scenarios to more complex environments (such as the curriculum-crate scenario or the full tournament map), we observed a case of - what we assume is - catastrophic forgetting. The model appeared to lose its previously acquired skills and struggled to perform tasks that were less demanding than those it had just learned.

This issue remained unresolved despite our best efforts.

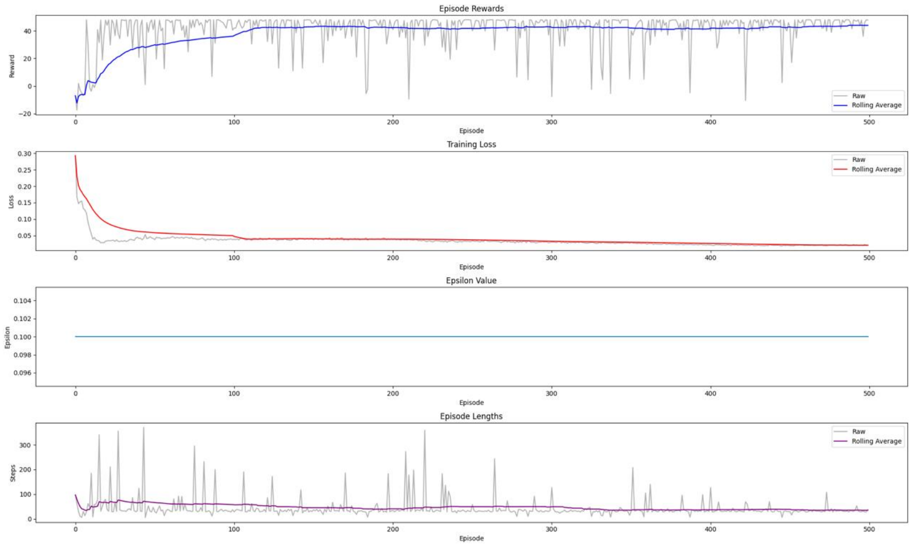

Figure 2: Results for curriculum coin training. Only few trainings episodes were needed, with a relatively small (and constant) ϵ of 0.1

2.3 Prioritized Experience Replay

Prioritized Experience Replay (PER) enhances our agent's learning by strategically sampling important experiences more frequently during training. Instead of uniform sampling, PER assigns priorities based on TD errors, effectively measuring how "surprising" or "unexpected" an experience was. This is particularly valuable in Bomberman, where critical moments like deaths, coin collection, and bomb encounters are rare but crucial for learning. These experiences typically generate larger TD errors, resulting in higher sampling priorities and more frequent replay. Our implementation uses a binary sum tree structure for efficient O(log n) priority-based sampling. New experiences initially receive maximum priority to ensure exploration, and their priorities are then updated based on their TD errors using the formula priority

$$(|TD\_error| + \varepsilon)^{\alpha} \tag{1}$$

To correct the bias introduced by non-uniform sampling, we use importance sampling weights. The system is tuned with α=0.6 for prioritization strength, β starting at 0.4 and annealing to 1.0 for bias correction, and ϵ=0.01 to ensure non-zero priorities.[1]

2.4 Model Improvement

After an initial size of the network with two hidden layers of size 64, we later changed the size of the first hidden layer to 128, since the number of inputs rose to 80. This change was made based on the heuristic described in the information bottleneck theory, which roughly states, that the representation space should first be expanded (80 → 128) and then compressed (128 → 64) for optimal deep learning. For further detail, one could use `https://arxiv.org/pdf/1503.02406` as a starting point.

Another big change we only made near the end of the project was the reduction of γ from 0.99 to 0.8. This resulted in much quicker and more stable learning on easier maps, but also to noticeable improvements on the classic tournament map. We presume this is the case, because in the TD update, the lower γ corresponds to a lower impact of the q-values and therefore a higher impact of the reward, which in turn makes reward shaping more influential to the training.

Another nob we tried to turn throughout the whole process was the epsilon value. In the end, our observations are the following: For smaller or simpler maps with a clear goal or a laid out path (like curriculum-coin), it seems like a small constant epsilon (like 0.1) leads to the most stable and fast learning, whereas on maps with a high degree of variability, a longer epsilon decay starting from a high value (like 0.9) going down to a low value (like 0.1) leads to a more broad understanding of possible situations, since the exploration phase is longer.

[1] The implementation and the hyperparameters were taken from the paper `https://arxiv.org/pdf/1511.05952`

We observed that training the last 1/4 of the time on the lower epsilon further helped refine the action selection for the beginning phase of an episode, after having learned goo q-values during the exploration phase.

2.5 Double DQN

We implemented a double DQN along the proposition in the Learnweb, but it did achieve worse results (see Figure 3) and took noticeably more time to train in comparison to a DQN. Therefore it was not reasonably trainable in the given time frame. Since DQN usually struggles with maximation bias, we thought double DQN could help. As Bomberman is an environment with relatively sparse positive rewards, there are more pressing issues and better opitmizations at hand that we decided to try. We dropped the approach because of these aforementioned reasons. Yet the code is still part of the repository.

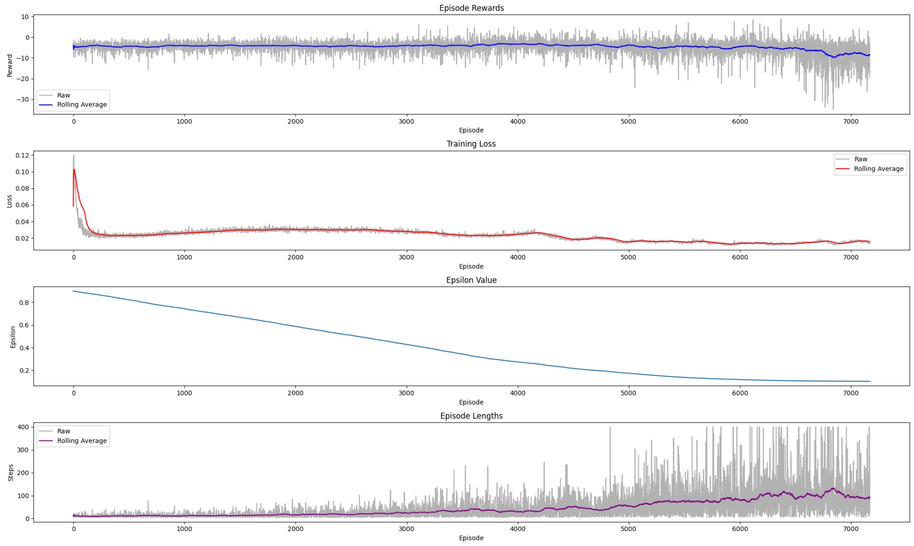

Figure 3: Results of Double DQN Training, plots for reward, training loss, epsilon decay, and episode length over 7000 Episodes

2.6 State Space Abstraction

After training the vanilla model on the full state space given by the environment, we could tell that the full state space was too large to efficiently train the agent. Therefore we tried to reduce the input by restricting our state space using a single padded 7x7 window of a „scope representation“ we had implemented for the rule-based-agent. The representation used breadth-first search to make a single array showing the different features like crates, coins or bombs with their own values each, and assigning a value to all the cells the agent could possibly move to without a blockade. This representation did not work very well though, since the network could not interpret the different values and the representation was not invariant with respect to the starting position. Our next approach worked better by using more relational inputs rather than a direct representation of the grid. This meant we used 8 inputs for the immediate surrounding cells, categorizing them into walkable and unwalkable, 16 inputs for the closest object in each direction, and 8 global features like the distance of the nearest coin, bomb and enemy for a total of 32 features. This change boosted performance significantly, while still remaining quite small in size. Our final enhanced representation extended this approach by adding 40 features for danger detection for all 8 directions, which included checking the bomb timer of each bomb in the surrounding cells. Some small changes were made to the existing abstraction, which resulted in a total of 80 input features. This once again boosted performance, especially in dodging bombs, but still remains faulty in certain cases.

2.7 State Space Transformation

2.7.1 Inspiration

- The agent's position in the map does not affect the possible actions.
- The agent's position in the map does not significantly affect the optimal action.
- Example:
  - The agent is in the top left quadrant.
  - The optimal action is to move to the right.
  - If the map was mirrored along its vertical axis (including the agent's position), the optimal action would therefore be to move to the left.
  - However, the state representation/observation is different in the mirrored case.
  - Even though the optimal action is virtually the same (just mirrored).
- Conclusion: If we can ensure that the agent is only learning on any one quadrant, and only observes states where it is in that same quadrant, variance is reduced.

2.7.2 Idea

- We transform the state space to only portray the agent in the top left quadrant:
  - If the agent is in any other quadrant, the matrix (numpy array) is flipped (mirrored) to place the agent in the top left quadrant.
  - If the agent walks across the halfway line either to the right or the bottom, the transformation is adapted dynamically: The agent “thinks” that it “bounced off” at the halfway line.
  - The actions are transformed accordingly: left ↔ right, up ↔ down.
- In theory, this should cut the state space by 3/4, without losing any information:
  - The whole state is still represented (just mirrored).
  - As there is no information tied specifically to the quadrant the agent is in, the transformation has no information loss.
  - Overfitting should not be an issue, as **all** input states have the agent in the top left quadrant (because all observed states are transformed before input).

2.7.3 Results

- Due to an implementation error that did not show up easily in debugging, the state transformation did not work correctly in the beginning, nullifying its results before the error was caught.
- In the few runs after fixing that error, no discernible effect could be seen. We believe this is due to multiple reasons:
  - Most of the refined state abstractions used do not take much of the original state arrays into account immediately, instead using only the immediate surroundings and then calculating relative positions.
    - While this should still be stabilised by the transformation (e.g. the relative distance of the next wall to the left is likely in a narrower range), the effect is not as significant as when using e.g. the entire arrays for walls, bombs, etc.
    - Input features such as the distance to the nearest coin is not at all affected by the transformation, as the distance between 2 points in the map is invariant to any transformations.
  - Even though all observations have the agent in the top left quadrant, this might still lead to overfitting in some way, especially with the more refined state abstractions and the relative positions.

2.8 Training Time and Computation Cost

The training duration is influenced by multiple factors, including processing power, episode length, and the computational device employed (GPU or CPU). Figure 4 illustrates a finetuning run comprising 9,500 episodes with relatively short episode lengths. This agent was trained on an NVIDIA 4080 Super GPU, which offers 16GB of memory. In an idle state, approximately 3.8 GB of this memory is utilized. The graph demonstrates that our model is not computationally intensive. The observed spikes in memory usage are attributed to various allocations initiated during the training process. The episode reward and episode length graphs support this thesis, as they do not exhibit data patterns that would require additional computational resources. The 9,500 episodes were completed in approximately 8,000 seconds, or 2 hours and 10 minutes. Other training runs, extending up to 50,000 episodes, required approximately 10 hours, which aligns proportionally with the time requirement observed in this case.

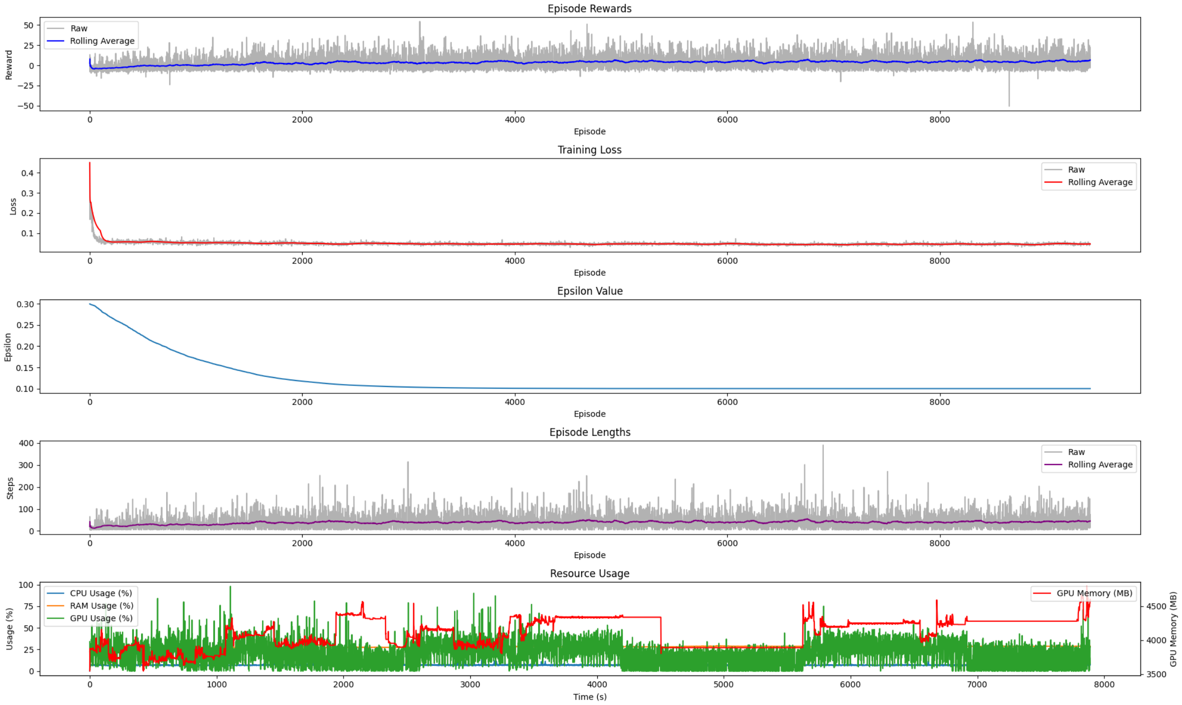

Figure 4: Ressource usage during training.

2.9 Best Performing Agent

Lastly, we want to show the the learning curve and stats of our best performing agent, which will compete in the tournament. The first training resulted in the data in Figure 5. We then finetuned the model and reiterated training, since the agent was a bit to aggressive and prioritized chasing rewards over its own safety. Finetuning results are shown in Figure 4.

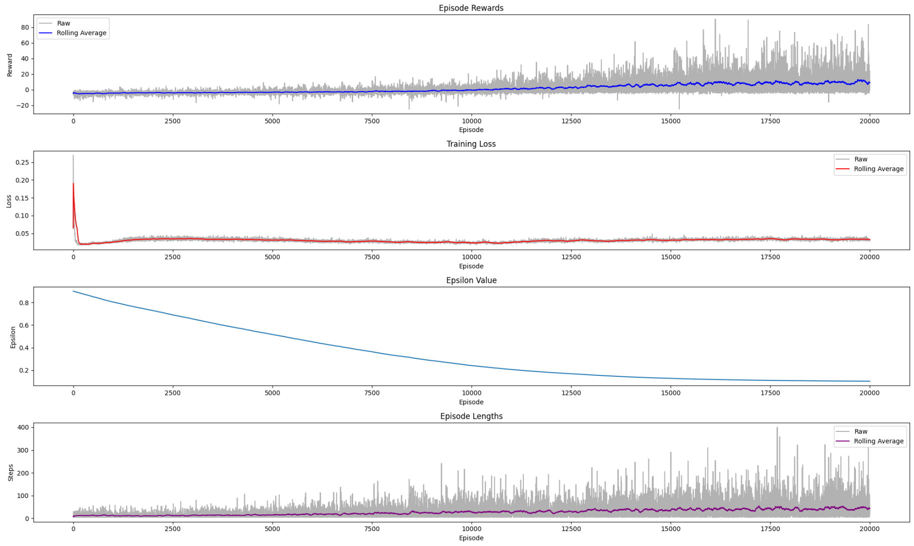

Figure 5: Learning curve of our best performing agent in full tournament space.

The code for this model is found in the folder "our_agent".