Report for Bomberman ML Project

by
Anu Reddy(3768482), Keerthan Ugrani(3770219), and
Florian Tichawa(vr281)

October 18, 2023

Contents

1 Introduction

The project task was to create and train a Machine Learning (ML) algorithm capable of playing the game "Bomberman". The program, referred to as "agent" from here on, should be able to run in the Python game environment provided by Prof. Koethe in his GitHub repository[3]. The environment provided takes care of the general game logic, calls all participating agents in order, and communicates the move returned by every agent with all participants. For the agent implementation, the main logic should be contained in the act() method of the callbacks.py file, as it will be the entry point for the game environment. In the end, every team should create at least two different ML models, one of which will be submitted to compete against the submitted model of every other team. The project code for our submission, including the entire coding history, is publicly accessible on GitHub[2].

1.1 Game Rules

In this version of Bomberman, all agents will be launched into an unknown arena consisting of rectangular grid built from the following tiles:

- Free tiles - these tiles can be moved through and have no further effects.
- Agents - these tiles contain an agent and can not be moved through.
- Bombs - these tiles indicate the position of a placed bomb amd can be moved through.
- Explosion tiles - these tiles destroy any agent contained within them, and spawn around a bomb four time steps after its placement.
- Solid tiles - these tiles can not be moved through.
- Crates - these tiles can not be moved through, but they will be destroyed by bombs and can contain coins.

- Coins - these tiles can be moved through and contain a coin which can be picked up by entering the tile.

Every agent will act in order, its actions will be resolved before the next agent acts, and the goal of the game is to survive longer than every other participant. In case of a tie (E.g. caused by two remaining agents dying to the same bomb), the total score will determine a winner. Taking out an opponent or collecting coins raises an agents score.

Furthermore, the game is limited to a total of 400 game steps, and every agent has to calculate its next move within 0.5 seconds. Failure to meet the time limit will be punished by not allowing the offending agent to move on this turn, and the time taken beyond the limit will be deducted from the agent's next turn timer. Illegal moves (E.g. attempting to move into a wall) will be caught by the environment and replaced with a WAIT action instead.

1.2 Agents Work-Flow

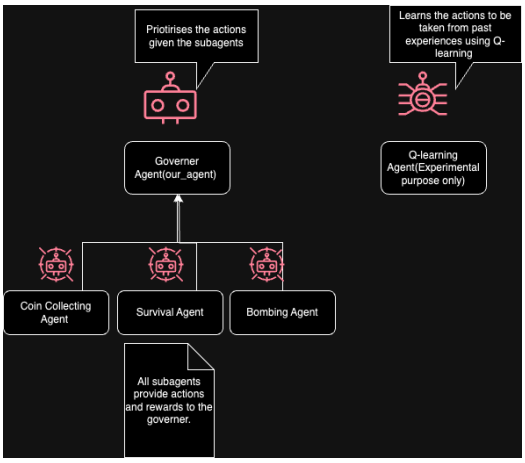

Figure 1: Our Agents Work-flow Diagram

2 Agent Design

To tackle the variety of different tasks our agent has to solve, we decided to split our agent into separate subtasks, each of which can be trained individually without affecting the performance of the other tasks. To combine all agents into a single agent capable of handling all tasks at once, we created a governor agent, whose only task is to handle the inputs of all subtasks, compare them, and pick the action provided by the agent with the highest confidence in its prediction. To enable this functionality, we extended the original agent structure with an additional next_move() method, which takes the exact same input as act(), but returns a tuple of the next move and a confidence value within [0, 1]. The governor then takes these move suggestions, weights them based on an internal decision system, and returns one of the provided suggestions as its overall choice.

The advantage of this approach is the ability to tweak the performance at one task without disrupting any other task - either by improving the corresponding sub-agent, or by adjusting the weights to accommodate for a hyperfocus on certain tasks.

For our submission, we decided to split our agent into the following sub-agents:

- Survivor: An agent focusing on not getting hit by explosions.
- Collector: An agent tasked with pathing to and collecting coins.
- Coin Creator: An agent trying to maximize the number of crates destroyed to generate new coins and paths.
- Pathfinder: An agent focusing on tracking down opponents.
- Bomber: An agent with the goal to take down opponents.

2.1 Governor Design

Florian

In the submitted version, we decided to go with a simple, static implementation of the governor. Every sub-agent is registered within the governor by providing its setup() and next_move() callbacks to the governor, which are then stored for later. During its own setup(), the governor forwards the call to all sub-agents, and then initializes the sub-agent weights for the decision algorithm. In its act() method, it once again forwards the call to all sub-agents, retrieves the suggestions and confidence values for each of them, and picks the move associated with the highest confidence value after multiplication with its stored weights. As there are up to six different moves an agent can return, our general goal was to aim for a confidence significantly higher than 0.16 for our sub-agents to differentiate between a random low-confidence and a deliberate high-confidence choice. The weights of all sub-agents were determined experimentally, and the last part of our "training" consisted of testing multiple variations, tweaking these weights whenever we saw the governor focusing too much on one aspect of the game.

2.2 Survivor Design

Florian

The task we considered to be the most important one is the Survivor agent. After some thought, we decided to only focus on short-term survival, which allows this agent to only use what we called "local vision", or vision of the 5x5 square centered around the agent. Since this limited input greatly reduces the complexity of the agent, we decided to use a three-layer neural network with 25 input neurons (Each of them corresponding to one tile of the local square), one hidden layer, and five output neurons, each of which encodes for one of the five possible actions (WAIT, UP, DOWN, LEFT, RIGHT).

Like every agent, the survivor also has to provide confidence for its prediction. The goal was to have high confidence (> 0.33, as there can be up to three different ways to escape an explosion) whenever the survivor is too close to an explosion, and a low confidence (< 0.2, as the agent has five different moves to choose from) if there is no immediate threat to our agent.

Fortunately, softmax and similar encodings already provide us with a bounded value in the exact range of our confidence values for every action, which made neural networks a great choice for the overall agent design we had in mind.

For the actual implementation of this sub-agent, we decided to use the pytorch library, which we had already been familiarized with during the lecture.

2.3 Bombing Design

Keerthan Ugrani

This task of the player is the deciding factor in winning the game. The bombing agent has a global vision of the game state, which means that this agent has information about the positions of the crates and the positions of the enemy players. The bombing agent has been programmed logically to strategically place the bomb in the game environment, this could include the case that the agent finds the dead end and finds out either of the obstacles is a crate. We also make this agent more sophisticated by effectively calculating the effectiveness of the bomb and taking steps to avoid getting caught in the explosion of the self-placed bombs.

The bombing agent provides confidence according to the score for all the possible moves, which means that the bombing agent returns the moves and the respective confidence of each move just like every other agent. Here it depends on the Q-value of each action taken into consideration and the move with the highest Q-value is returned.

Q-learning is a foundational reinforcement learning algorithm used in machine learning to find the optimal action-selection policy for a given finite Markov decision process. The algorithm iteratively refines estimates of action values, allowing an agent to make informed decisions in an environment. The Q-Learning update equation is a crucial component of this algorithm, determining how the agent updates its knowledge based on experiences.

The Q-Learning update equation is as follows:

$$Q(s,a) \leftarrow (1-\alpha) \cdot Q(s,a) + \alpha \cdot \left(r + \gamma \cdot \max_{a'} Q(s',a')\right)$$

The update equation combines the current estimate Q(s, a) with new information obtained during an agent's interaction with the environment.

The Q-learning functions collectively form a comprehensive toolkit for training and evaluating an agent in a game environment. By addressing reward assignment, policy generation, experience accumulation, and neural network updates, these functions contribute to the agent's ability to learn effective strategies for gameplay.

2.4 Coin Collecting Design

Anu Reddy

Reinforcement Learning (RL) principles are used in the coin-collecting agent, specifically a Deep Q-Learning algorithm [1]. A Q-Network, a form of neural network used to approximate Q-values, provides the underlying structure. As a result of starting from a state and taking an action, these values indicate the quality of the action, which represents how many predicted rewards an agent is likely to receive after completing. An architecture consisting of three layers of feed-forward neural networks is used.

In the context of coin collection, states represent the various configurations of the game board at any given time step, including the locations of coins, obstacles, and the agent. The actions are the possible moves the agent

can make (up, down, left, right). The Q-values generated by the Q-Network guide the agent in choosing actions that maximize future rewards.

Experiencing and exploiting are balanced by epsilon-greedy decision-making. Initially, the agent will take random actions to explore its environment because of its high epsilon. Over time, epsilon decays, and the agent increasingly relies on Q-values to make decisions, transitioning from exploration to exploitation.

When playing the coin-collecting game, the objective of the player is to efficiently explore the game grid, gather coins while avoiding obstacles. The game board configuration is represented by numerical values that are saved as states and fed into the Q-Network to predict Q-values for potential actions. Afterwards, the epsilon-greedy policy is utilized to select actions.

To sum up, the coin-collecting agent has a well-structured implementation of the Deep Q-Learning algorithm. Its purpose is to learn how to efficiently collect coins while navigating through complex situations over time, by adapting and optimizing its policies. This agent serves as a good model for RL problems due to the integration of experience replay and the epsilon-greedy strategy. Below is the screenshot of how the initial coin-collecting agent works. Later bombing logic(designed by Keerthan Ugrani)was also integrated with coin collecting and was working as one agent, and it was governed by the governer agent(designed by Florian), to work as a complete agent performing all the tasks in the game).

2.5 Second Agent

Anu Reddy

This agent was solely built for experimental purposes, wanted to explore how the agent would work when completely implemented in Q-learning. The agent is built with a Deep Q-Learning network and operates in a Bomberman-like environment. It is tasked with navigating the terrain, gathering coins,

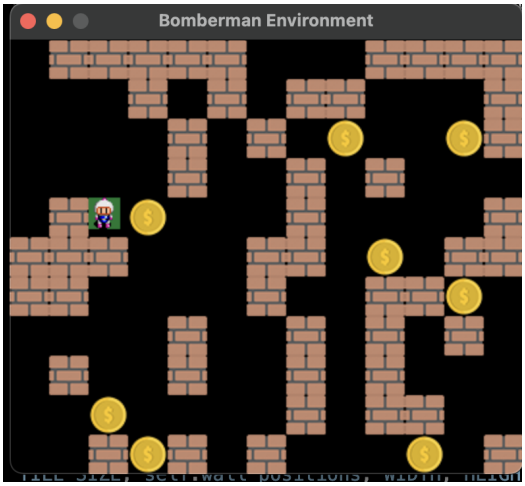

Figure 2: Intial Coin Collecting Agent

avoiding or destroying adversaries, and overcoming obstacles such as walls and crates. A neural network that has been trained to anticipate the Q-values for each conceivable action given a state determines the agent's behavior, and it chooses actions based on these predictions.

The Player class represents the agent and is initialized with its position, speed, and environment information such as the positions of walls, crates, coins, and Enemies. It has the ability to travel in four directions and drop explosives. The move method modifies the player's location based on the action specified while ensuring that the new position is not a wall or crate and that it is inside the environment constraints. The place-bomb method adds a timed bomb to the player's list of active bombs.

The update-bombs method handles the logic of the explosives. Each bomb's timer is depleted on each step until it hits zero, at which point the bomb explodes. The explosion destroys crates and perhaps kills the player if they are within a cross-shaped area. The method returns whether or not the player was killed and which crates were destroyed. If the player collects a coin, the coin is removed and the reward is increased. The player's bombs are also updated in each step, with rewards adjusted based on the destruction

of crates or the player's death due to their own bomb.

The DQNSolver class is a neural network model with three layers that assists the agent in learning. The act technique determines the agent's actions. To encourage exploration, the agent randomly takes an action with a probability equal to the exploration rate. Otherwise, it selects the action with the highest expected Q-value based on the neural network, which exploits learned behaviors.

The remember technique saves the agent's experiences in a memory deque for further training, including state, action, reward, next state, and if the game is completed. The experience-replay function [1] uses a batch from memory to train the neural network. The Q-values are updated to the target Q-values, which are calculated using the received incentives and the expected Q-values of the following states. Over time, the exploration rate decreases, decreasing the likelihood of random acts and increasing reliance on learned behaviors.

3 Training

Due to the subdivided structure of our agent, we had to entirely separate the training of our sub-agents. For each agent, we defined a basic metric we wanted to measure its performance on, agreed on a method to calculate the confidence to keep it as consistent as possible between all sub-agents, and decide on the best training method for all of them. For the actual training session, we used our private computers, as well as a Linux web server owned by Florian to allow for longer runs without having to occupy our private devices for an extended period of time.

3.1 Governor Training

Florian

As already mentioned in chapter 2.1, the governor was not using any ML algorithms, but rather consisted of static, handpicked parameters we determined by trial and error after all sub-agents had finished their training. We started out with equal weights but soon realized that our agent was putting too much emphasis on survival, and the survivor sub-agent was never allowing any other sub-agents to act - which did result in our agent surviving until the end, but often losing after 400 turns due to the point tie-break.

3.2 Survivor Training

Florian

For the survivor agent, we started out by deciding on an ideal localized survival strategy. After some tests, our solution came out to be fairly similar to the one present in the rule-based agent provided by the environment: Our agent should avoid stepping into explosion tiles or the cross-shaped dangerous area surrounding any bomb, and if caught inside of one, it should try to escape to the nearest free space, prioritizing turns around a corner over simply moving away from the bomb. There are some edge cases where this algorithm fails, for example, bombs reaching all the way into a dead end with no free tiles to escape to - but because our agent was only supposed to have a localized vision, these cases would be outside its vision range and thus not be detectable anyway.

As a target metric, we initially chose the number of survived turns, but soon discovered that the algorithm was not converging very quickly. After trying the number of survived bombs as a training metric and still not getting any satisfying results, we decided to change our approach altogether: Because our goal was to mimic an algorithm we already had, and not to find a solution

to a problem we hadn't solved yet, we didn't need to define a target metric. Instead, we did exactly what we actually wanted our network to do: Every time a move was requested by the environment, we ran our existing algorithm on the same input in parallel, determined the "ideal move", and stored this move as the target label for our weight adjustment later on. This changed approach greatly improved the training runtime because we were now able to clearly label every single move taken as "good" or "bad" in a vacuum, instead of having to analyze the entire game to determine the quality of every move taken throughout it.

In addition, we were now able to replace our default training environment, the classic arena with three rule-based agents, with a much smaller arena of 7x7 tiles (Excluding the border) and no agents, because the network simplifies anything but bombs, free tiles, and explosions to solid tiles. Another issue we noticed with our training was how the survivor agent barely had any reasons to act for the move of the time, which resulted in the network heavily favoring the "WAIT" action over anything else. With the smaller environment and the missing agents, we had to somehow introduce a new source of bombs anyway, so instead of agents placing bombs, we randomly placed bombs on free tiles, which allowed us to speed up training even more and even solved the lethargy issue of our network.

3.3 Bombing Training

Keerthan Ugrani

For the bombing agent, we understood the logical way of implementing the bomber agent and then converting it to features using a tensor, so that we could feed it to the model as the input for training purposes. In the training process, we calculate the total reward based on a set of predefined events that occurred during a game step. Each event type is associated with a specific reward value. We also provide a clear mapping logic between game

Figure 3: Training progress

events and their corresponding rewards, enabling the agent to learn from its interactions with the environment. We incentivize desirable behaviors (e.g., collecting coins, killing opponents) and penalize undesirable ones.

We also introduce a strategy for the agent to decide whether to choose actions based on the current knowledge (exploitation) or to explore new actions (exploration)

We keep checking on game events occurring and the end of each round. To handle the accumulation of experiences, train the network, update the network, and track game scores. These functions ensure that the agent learns from its interactions with the environment.

As a target metric, we initialized the highest rewards for killing the opponent and then exploding the crates. The agent when it reaches the dead-end corner and if either one is a crate then it places the bomb and earns the reward. Here we not only gave rewards for bombing up the opponent but also for collecting the coins placed randomly in the custom environment so that the agent learns to collect the coin after bombing up the crate.

3.4 Coin Collecting Training

Anu Reddy

The coin-collecting agent undergoes iterative training through a series of 1000 episodes, where each episode represents a complete game from start to

finish(refer the to Figure 4). During each episode, the agent interacts with the environment by making decisions, receiving rewards as feedback, and adjusting its strategy accordingly. This process is supported by the Deep Q-Learning algorithm, which is a variant of Q-Learning that utilizes neural networks and experience replay.

At first, the agent explores the environment by taking random actions, thanks to a high epsilon value in the epsilon-greedy policy. The epsilon parameter determines how much exploration versus exploitation is prioritized. A high epsilon promotes exploration, which helps the agent discover and learn the various states and rewards in the environment. As time goes on, epsilon decreases, shifting the balance towards exploitation, where the agent relies more on its acquired knowledge to make decisions.

During each episode, the agent observes the current state of the environment and decides on an action based on its policy. The policy is derived from the Q-values generated by the Q-Network, which is a neural network that approximates the Q-value function. Once the action is selected, it's executed in the environment, leading to a reward and a subsequent state. This transition is stored in the agent's memory, which holds a collection of such transitions for experience replay.

Experience replay is essential in stabilizing the training process. It involves randomly sampling a batch of transitions from the agent's memory and using them to update the Q-Network. Random sampling helps to break the correlation between consecutive transitions, leading to a more stable and efficient learning process. Each sampled transition includes the state, action, reward, next state, and the done flag, which indicates whether the episode has ended.

To update the Q-Network, we minimize the loss between predicted Q-values and target Q-values. To calculate the target Q-value for a given state-action pair, we add the immediate reward to the discounted maximum Q-value of the subsequent state, computed using a temporal difference estimate.

The Adam optimizer facilitates backpropagation, which adjusts the weights of the Q-Network to minimize the loss and learn the optimal policy over time.

We typically monitor the agent's performance by the total rewards accumulated in each episode. These rewards are often plotted to visualize the learning progression and gain insights into the effectiveness of the training process. We can also save the trained model periodically to capture the evolving Q-value function, providing checkpoints for analysis, evaluation, or deployment in similar environments.

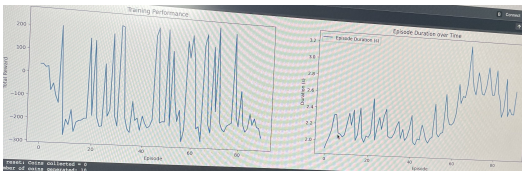

Figure 4: Reward trend and Episode duration during Training progress over every 100 episodes

From the graph, The agent is learning to collect more coins over time, but this is coming at the cost of longer episode duration. The agent's performance is still improving, but it is plateauing at around 40 coins per episode. This suggests that the agent may be approaching the maximum number of coins that it can collect in the environment. The agent's performance is also noisy, with some episodes resulting in significantly more or fewer coins collected than others. This suggests that the agent is still learning and has not yet fully optimized its policy.

3.5 Q-learning Agent(Experimental Purpose)

Anu Reddy

Deep Q-Learning is used to train the agent over several episodes. The agent acts in each episode depending on its current policy, which is a mix of exploration and exploitation guided by an exploration rate. Actions are either

chosen at random to explore the environment or chosen for the highest projected Q-value in order to leverage learned knowledge. Each action results in a different state and a matching reward. These data points for state, action, reward, and next state are saved in a memory structure for Experience Replay.Periodically, batches of these stored experiences are randomly sampled and used to train the neural network. The network is trained to predict Q-values that represent the expected future rewards for taking specific actions in given states.

The Q-values are updated using the Q-learning update algorithm, with the goal of reducing the mean squared error between the current Q-values and the updated values, which contain the reward and the discounted maximum predicted Q-value of the next state. The exploration rate decreases across episodes, enabling the agent to gradually move from exploration to exploitation.

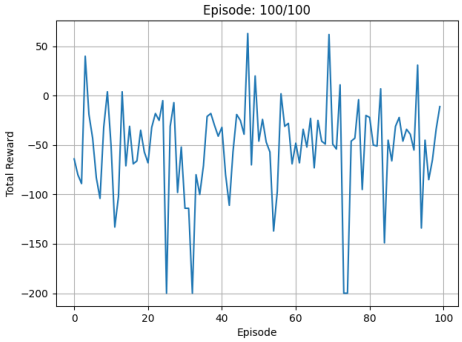

Figure 5: Reward trend during Training progress over 100 episodes

The reward graph demonstrates that the agent can learn from its mistakes. The agent receives tiny rewards in the early episodes. However, when the agent becomes improved at avoiding bombs and blast crates, its rewards begin to rise. It also means that the agent can learn from its errors and improve its performance as time passes. The reward graph demonstrates that the agent can extrapolate its knowledge to new events. The agent is able to

collect rewards in all episodes, despite the fact that the environment changes. This implies that the agent can apply its knowledge to new scenarios.

In a nutshell the agent learns by continuously interacting with its environment, gathering experiences, and enhancing its neural network to better predicted future rewards, leading to improved action selection over time.

4 Discussion

In our opinion, the governor approach we took for our primary submission has a lot of potential. We already pointed out some of its advantages over a monolithic implementation in chapter 2, but we would also show some of its flaws here, and how we tried to avoid them:

First and foremost, the governor only works as the sum of all of its parts. If one part is broken or missing, achieving meaningful results is nearly impossible. In our case, the central element of Bomberman was survival: Without a surviving agent, we can't use bombs, move around, collect coins, or do anything the other sub-agents are supposed to do. But this also goes for other synergies: There is no point in collecting coins if we can't create any, and there is no point in finding a path to the nearest opponent if there are crates blocking the way, making it impossible to move. To circumvent this issue, we took two different approaches: In some cases, like the survivor, we created a static, rule-based alternative for the ML agent, who would stand in for the actual agent until it was ready, allowing us to train everything in parallel without having to wait for the other sub-tasks to complete (Which would defeat one of the main goals of this approach). In other cases, like the coin collection and bombing agents, we simply decided to merge these two tasks, as they were too closely related to effectively split them up, and train them together as a single sub-agent. This approach does go against the general concept of our agent design, but we decided to take these steps in favor of time and simplicity. With more time at our hands, we probably

could have introduced even more tasks and split up the existing ones even further to allow for a more fine-grained control over our agent.

If split up sufficiently, this approach could easily be used to customize the agent for whatever purpose it should fulfill: Increase the weight of a bombing sub-agent to increase the agent's level of aggression, decrease the priority of the survivor to allow for riskier maneuvers, increase the priority of collecting coins over destroying opponents, etc.

We could even take this concept even further and change our weights (Which can be seen as meta parameters for our model) to change over the course of a game depending on certain factors like a number of opponents, our current standing, etc. Finally, the governor itself could be implemented as some sort of simple ML algorithm, with its weights being the parameters adjusted based on training data. Based on our experience with our agent training so far though, it would most likely require an unjustifiable amount of training data to even get decent results for such a complex agent - even if we assume the sub-agents to be fully trained already.

Another approach to improve the governor would be to replace the decision algorithm: Instead of simply picking the action with the highest confidence, the governor could use a decision-forest-like approach of letting all sub-agents "vote" on the best action. For example, if one agent suggests the move "WAIT" with high confidence, but all other agents suggest "LEFT" with slightly lower confidence, then choosing the latter action would be beneficial for multiple goals, and thus most likely be more beneficial overall. An algorithm combining all suggestions in a weighted vote system and picking whatever action the most sub-agents vote for improving the governor's performance without requiring any additional training. On the other hand, this approach would most likely not be fit for the training phase, since it makes decisions significantly harder to propagate backward to their original sources.

Coming to the 2nd Model, we thought of completely working agent that could comply with all the tasks as one agent. Combining the logic for all the

primary tasks was a challenge for the agent to learn efficiently and moreover, the agent was not much of exploiting rather than exploring, due to lack of computational resources, this agent was not chosen for the competition. Here are the some of improvements that could have been incorporated.The agent's performance can be increased further by combining multi-agent learning for more adaptability. Transfer learning could speed up training by incorporating knowledge from previously learned models. Implementing a more sophisticated memory system, such as prioritized experience replay, can help to improve learning efficiency. By including a bigger and more comprehensive field of vision of the gaming environment, the agent's perspective can be expanded.

References

[1] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[2] *Nightara/Bomberman GitHub repository*. Accessed on 2023-09-20. URL: https://github.com/Nightara/Bomberman.

[3] *ukoethe/bomberman$_r$lGitHubrepository*. Accessed on 2023-09-20. URL: https://github.com/ukoethe/bomberman_rl.