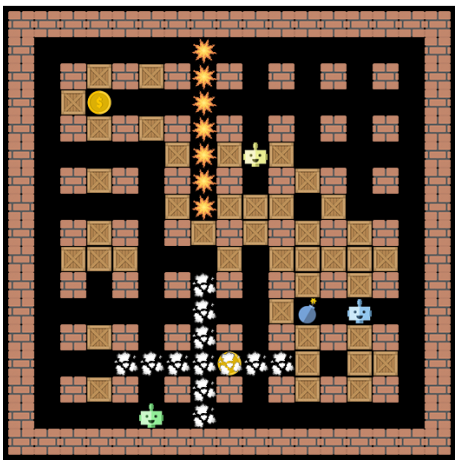
Final Project Report

Machine Learning Essentials Summer 2023

Reinforcement Learning for Bomberman Game Agent



Team Name:	Least Misérables
Authors:	Hammad Aamer, Shivangi Sharma
Date :	October 2, 2023
GitHub Repository URL:	madham97/Least_Miserables

GitHub link: http://github.com/madham97/Least_Miserables

Contents

1. Introduction

1.1 Problem Statement

Bomberman is a classic grid-based strategic game that has been played across the world for decades. The objective of this project is to train an agent that is capable of making real-time decisions to play the Bomberman game, using Machine Learning techniques, against other trained agents as well as rule-based game agents.

1.2 Game Setup

In order to develop strategies that allow an agent to play Bomberman in a demanding multi-agent environment, we start by understanding the game environment.

The bomberman playing field for our project is a 17x17 grid arena. Each block in the grid is occupied by one of 4 objects: a WALL, a CRATE, a BOMB, an AGENT, or a COIN. Following is a description of each of these objects' functions individually.

AGENT: An agent is the primary object that plays the game. The goal of an agent is to collect coins and eliminate opposing agents while avoiding getting eliminated themselves. To this effect, based on the state of the game an agent is able to make one of 6 actions per each step of the game; namely “UP”, “DOWN”, “LEFT”, “RIGHT”, “BOMB”, and “WAIT”.

BOMB: A bomb is an object dropped by an agent in a block when it performs the “BOMB” action. The bomb is dropped at the position of the agent that drops it. This means that upon dropping the bomb, an agent and a bomb occupy the same block of the arena. The function of bombs in the game is to destroy crates and agents. Each bomb has an explosion timer after which it releases an explosion. In our game setup, an explosion impacts the block where the bomb is placed as well as 3 other blocks in each direction. The explosion lingers for 2 turns, the turn it explodes and one more turn. An explosion will destroy any agent or crate

that occupies the same block and can only be stopped from reaching further if a wall is in the way.

WALL: A wall in an indestructible, untraversable object. This means that it can not be destroyed by a bomb left by an agent, nor can an agent move into a block occupied by a wall. The purpose of walls in the game is to limit an agent's maneuverability and force an agent to take clever actions to work around the obstacle.

CRATE: Crates are destructible, untraversable objects in the game. A crate serves to randomly hold coins inside itself and will only release the coin if it is destroyed by a bomb dropped by an agent.

COIN: A coin introduces one of the primary goals of our agents. A coin will occupy the block of the crate that held it once the crate is destroyed. An agent aims to collect as many coins as possible as collecting a coin rewards the agents with a point.

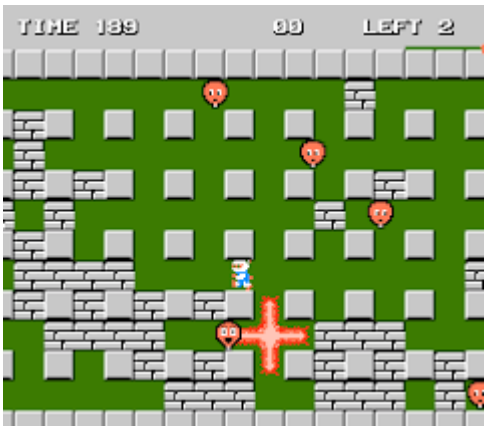


Image 1: Screenshot of Bomberman 1983

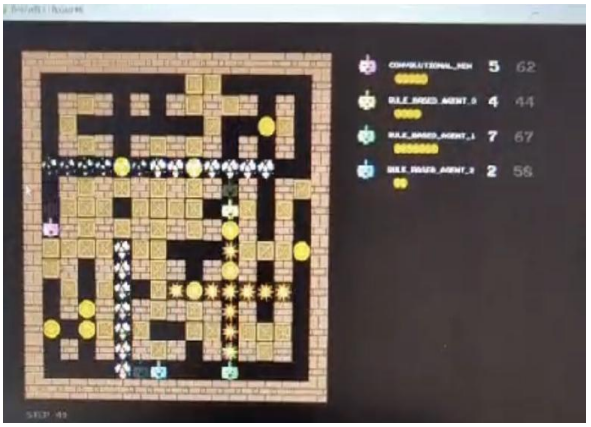


Image 2: Screenshot of the submission Bomberman

In a round of our game, four agents perform actions in each turn. Each agent aims to collect coins and eliminate other agents. Collecting a coin rewards the agent 1 point while killing an enemy agent rewards 5 points. A turn is a time step in the game where agents can perform any one action. Crates can be cleared with well-placed bombs and will occasionally drop coins that can be collected for points. However, the primary criterion for the final score is to blow up opposing agents while avoiding being blown up. The game ends when either all agents are eliminated or when 400 turns have taken place. At that

point, the agent with the most points accumulated is the winner of that game round.

1.3 Approaches to Solve this Problem

A Bomberman gaming agent can be trained using a variety of machine learning methodologies. Here are some examples of possible ML strategies:

- Reinforcement Learning (RL) using Q-Learning, Deep Q-Networks (DQN) and/or Policy Gradients.
- Imitation Learning using Behavioural Cloning or Inverse Reinforcement Learning (IRL)
- Deep Learning Techniques like Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs)
- Monte Carlo Methods etc.

2. Methods and Techniques

In this section, we describe the learning methods used to create our Bomberman-playing agents. We detail the crucial design choices that went into the development of these agents as well as define how we created features for our agents to observe the game arena. We will be going over both techniques we were satisfied with, as well as techniques that we attempted but abandoned.

2.1 Features

2.1.1 Linear features (Shivangi)

Our first approach to constructing our features was a linear one. We took our 17x17 game arena and converted it into a matrix with different values for each of the game objects and then we flattened it into a vector of size 289. Zeroes in the vector would signal free space.

This was a very experimental approach where we were laying the foundation of our pipeline. Soon enough we abandoned this method as we realized that our models were not being able to make sense of what each encoding signified. Even after significant epochs the agent would attempt to walk into walls and

not pick up coins. According to our interpretation of this performance, we concluded that the model is most likely taking the encoding as scores and not objects, hence the inability to form a discrimination.

The second version of our linear feature construction had us trying to practice a version of one-hot-encoding where our values would only be either zero or one. To this effect, we took our 17x17 game arena and added channels to it with each channel corresponding to a particular object in the game and a value of 1 signifying the presence of that object at that position. We ended with a total of 7 channels corresponding to our agent, crates, walls, coins, bombs, explosion, and enemy agents resulting in a 7x17x17 matrix flattened into a 2023 size vector. We tested this version of our feature as well but we soon realized that this feature vector contained a lot of redundant data. For example, the position of the walls will always be constant yet they are adding a huge amount of data to our vector. Furthermore, the vector becomes increasingly sparse as the round progresses resulting in proportionally less and less useful data for our agent to effectively train on. Conclusively, while this method was able to reduce overall loss a bit better than the previous approach, the actual results were very similar and as such we did not move forward with linear features.

2.1.2 Convolutional features (Hammad)

Building a convolutional feature matrix was something we were more hopeful about. This was primarily due to the fact that convolutional networks are very powerful in understanding the relations between neighboring indices in a feature matrix; a property that is very valuable in playing a game like Bomberman.

For our first implementation of a convolutional feature matrix, we simply used the same principle as our original linear feature implementation and encoded each block with a value corresponding to a particular in game object. In this iteration, we further added another integer for the situation where an agent and bomb occupy the same block; that is right when a bomb is dropped. Once again, while the drop in loss was much more rapid with this convolutional feature compared to the linear feature, the actual performance would barely show any improvement even after a significant number of epochs.

Our second attempt at utilizing convolutional features was by utilizing the same one-hot-encoding technique as the linear version, but not flattening the matrix. The results from such a feature input were significantly better than any of the implementations so far. Not only was the loss curve steeper, but the performance of the agent in the game was much more impressive as well.

Building on the success of our convolutional feature we made attempts to improve upon it. As discussed previously, this version of encoding the game arena leaves a lot of redundancy and does not make for effective training. For example, in the input channel that contains enemy agents, there is only a maximum of 3 indices filled out of 289. To try and remedy this, we decided to experiment with reducing the view of the agent. To that effect, we took the same 7x17x17 matrix and trimmed it down to a 6x11x11 matrix by taking only the 5 neighboring blocks of our agent in all directions as well as removing the channel that stored our agent's location. This approach meant that we also had to introduce padding to our game arena so that an appropriately sized matrix can be created when the agent is on the edges of the arena. Furthermore, we experimented with different ranges of how far the agent can observe until settling on 5. This feature had by far the best performance of any we had constructed yet and we decided to continue with it throughout the project.

2.2 Models

Our primary approach involves implementing two different machine learning techniques: an Imitation agent and a Deep Q-Network (DQN) agent.

2.2.1 Imitation Learning (Shivangi + Hammad)

Imitation learning allows an agent to learn from human or expert demonstrations, in our case the already existing rule-based agent present in our Bomberman environment, thus resulting in a solid initial policy and following advantages:

- It ensures that the agent begins with a policy that produces suitable and safe behaviour.
- It protects the agent from exploring risky actions during the early phases of training.

- It also speeds the learning process by using existing expertise, thus allowing the agent to significantly reduce the time required to train.

Utilizing the previously explained feature matrix we developed a convolutional model for the imitation learning. The following image shows the structure for our model as well as other specifications.

Optimizer: Adam
Loss Algorithm: Cross Entropy
Learning rate: 0.0001
Training device: Cuda
Inference device: CPU
Kernel size: 3x3
Stride: 1
Padding: 0

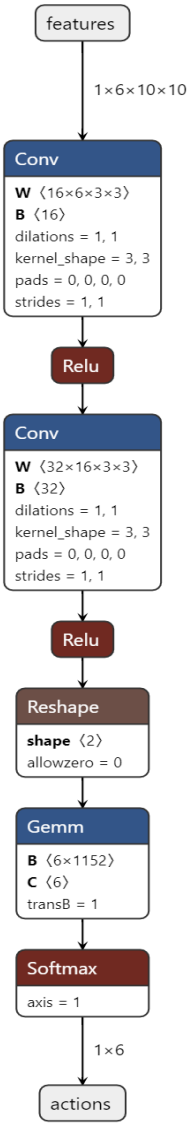


Image 3: Neural Network

2.2.2 DQN Learning (Hammad)

Deep Q-Networks (DQNs) offer a powerful framework for training agents in Bomberman. DQNs excel at learning optimal action-value functions, which are crucial for making strategic decisions in complex, grid-based games like Bomberman. Some of the key advantages of DQNs include

- Their ability to learn from interactions with the environment, gradually improving their policies through exploration. This self-learning aspect ensures that the agent can adapt and refine its strategies over time.
- DQNs can handle both discrete and continuous action spaces, making them versatile for the diverse actions required in Bomberman.
- DQNs are known for their ability to generalize from learned experiences, which is essential for making informed decisions in novel game situations.

These characteristics make DQNs a suitable choice for developing Bomberman-playing agents that can learn effective strategies, adapt to changing game states, and ultimately outperform predefined rule-based agents.

For DQN learning we used the same structure as the imitation learning model with a few key parameter changes and additions:

Optimizer: Adam
Loss Algorithm: Smooth L1 Loss
Learning rate: 0.000001
Training device: Cuda
Inference device: CPU
Kernel size: 3x3
Stride: 1
Padding: 0
Gamma: 0.999
Buffer size: 1000
Batch size: 64
Epsilon: 0.3

In the mentioned parameters, the most striking change from imitation learning is the reduction of the learning rate from 0.0001 to 0.000001. This is owed to a very critical discovery that we came across through our experiments. We

realized that DQN learning is much more sensitive to the learning rate than imitation learning. For a significant period we were unable to understand why the DQN was not able to produce a similar loss decay to our imitation model. This is until we changed our learning rate at which point we started observing more promising signs.

2.2.2 Ensemble modeling (Hammad)

Ensemble modeling techniques offer a potent approach to enhancing the performance and robustness of reinforcement learning agents in the challenging domain of Bomberman. By combining multiple learning algorithms or models, ensemble methods provide several key advantages such as

- Mitigating the risk of overfitting, as diverse models with complementary strengths and weaknesses can compensate for each other's limitations.
- Encouraging diversity in the exploration of strategies, increasing the chances of discovering effective, yet unconventional, gameplay tactics.
- Improving the generalization of learned policies, enabling the agent to adapt more seamlessly to the dynamic and evolving nature of the Bomberman game.

Overall, the application of ensemble modeling holds promise for creating highly adaptable and strategic Bomberman-playing agents capable of outperforming predefined baselines and adapting to various in-game challenges.

In testing out a number of different training techniques, we found that particular agents perform better in particular situations. This difference in capabilities was the motivation in including ensembling for our final agent. We chose 2 differently trained agents each with their own strengths and have each perform inference on the same environment. We observed that when softmax output of our main model is spread out between actions in one model, the other model often produces an output that is closer to the optimal solution. As such we assign weights to the predictions of each model and to get our final action we do an argmax of the weighted sum of the two predictions. We will explain more regarding the behaviour and training techniques used for each agent in the Training section. For our particular deployment, we chose a weight of 1 for our main model and 0.2 for our less important model.

3. Training

For the training of our models we deployed a variety of techniques throughout our development process. In this section we will go over these techniques and explain why we decided to use or not use each.

3.1 Settings based training

In our training process we modified our game configurations away from the classic one to drive training in our desired direction as well as avoid falling into local gradient traps.

3.1.1 Scenario tuning (Hammad)

One of the notable training choices we made was to provide our agent with increasingly complex goals so that it can iteratively learn the best action. We started with the simplest setting possible being a game arena with no crates and enemies and a large number of coins. In this iteration we expected our agent to be able to mimic the rule-based agent and realize that it is aiming to collect the coins scattered across the arena. Our agent was able to learn this behaviour soon enough and collect coins efficiently.

The next stage of our iterative approach was to introduce crates that hid coins. This is the first time our agent is forced to drop bombs for the purpose of destroying crates and with it comes the first major hurdle of our agent as well: not killing itself by a dropped bomb's explosion. This would go on to become our biggest challenge throughout development as survivability is the most primary function of our agent. It took our agent substantial epochs to survive the first few turns of the game as even though it would realize it needs to place a bomb to progress, it would not realize that it is in the blast radius.

Furthermore, with an explosion time of 2, even when our agent would avoid the initial explosion, it would most often walk into the lingering explosion and kill itself. For this purpose we modified the explosion channel of our feature matrix so that it would retain memory for the last 2 turns as well. In this manner, we would be informing our agent of danger from an explosion for 3 turns. This would go on to be very useful information for our agent and we saw a visible improvement in performance as a result. The third stage of our

scenario training was the introduction of enemy rule-based agents. While at this point our trained agent was decent at avoiding explosions, the main objective to learn in this iteration was to target agents alongside crates as well. The location of enemy agents is embedded in a channel of the feature matrix. In training, It was not very difficult for our agent to start targeting enemy agents but once again, with the introduction of more agents came more bombs and survivability came into play. We also weaved in epochs where there were no crates present so that our agent could realize faster that it needs to target enemy agents too. The reason for not training solely on this no crate setting was to ensure that our agent does not forget that destroying crates is also an important aspect on how to play the game.

3.1.2 Environment tuning (Shivangi)

Through our experimentations we observed that the agent has particular difficulty to survive when it is in cramped areas filled with crates. As such, the beginning of the stage is most often the hardest period for the agent to survive and overcome whereas it becomes easier later on in the game if the agent survives that long. Noticing this, we decided to limit our training games to a maximum of 50 turns before ending the round. This was particularly deployed when training in the second iteration of our previously explained scenario tuning, which is a game arena with only crates and no enemy agents. The reason for that is while we are not very interested in the later part of the game when there are fewer crates and more free spaces, we are definitely interested in later turns when there are still more crates and enemy agents present to attack. Hence this method was only limited to that scenario training iteration. The result of this selective training was much faster and more impactful epochs in our training.

3.2 Pipeline based training

In this subsection we will discuss how we utilized our 2 modeling strategies, namely imitation learning and DQN learning to train our agent.

3.2.1 Imitation Learning (Shivangi + Hammad)

Our Imitation Learning approach was by far the best technique to get our agent to reasonable intelligence. We carried out imitation learning in each turn of the

game where we created the feature vector based on the game state and fed it to our model as well as the rule based action function. On getting these two predictions we would calculate the cross entropy loss between them and use it to improve our model using back propagation. We would make sure to avoid any redundancy, such as remaking the feature vector for the two predictions, to make epochs as fast as possible.

3.2.2 DQN Learning (Hammad)

The first step of DQN Learning was to configure our reward criteria. The game environment already had some very useful event logging capabilities that we would also go on to use for constructing our rewards. Below is the list of rewardable events that we utilized in any stage of training:

- COIN_COLLECTED
- COIN_FOUND
- KILLED_OPPONENT
- KILLED_SELF
- MOVED_DOWN
- MOVED_UP
- MOVED_RIGHT
- MOVED_LEFT
- BOMB_DROPPED
- INVALID_ACTION
- WAITED
- CRATE_DESTROYED
- SURVIVED_ROUND
- GOT_KILLED
- CLOSER_TO_ENEMY
- FURTHER_FROM_ENEMY
- DANGER_ZONE
- SAFE_ZONE

The above events include both good ones such as KILLED_OPPONENT, as well as bad ones such as KILLED_SELF.

Of the events mentioned, not all were used by us in the final agent submission as we realized that they do not intuitively make sense with our training approaches. Features CLOSER_TO_ENEMY and FURTHER_FROM_ENEMY were developed to signal to the agent whether it has moved further or closed to the

enemy based on the change in mean square distance between itself and the closest enemy agent after taking an action. We decided to remove these events as we realized that with a limited view of the game arena, it is not reasonable for us to expect our agent to act on the reward and move closer to the enemy if the enemy is outside our agent's field of view. We considered adding direction information to our features as well so that our agent can know in which direction the closest enemy is, but ultimately we decided to scrap the idea and have our agent give priority to what is already closest to it; especially because self killing was such an issue already for our agent.

Choosing reward amounts for each event was ultimately the most difficult aspect of DQN Learning. Owing to our iterative scenario based training process we realized that it does not make much sense to have the same rewards for events in each stage of training. For example, in the stage where there are no enemies or crates, dropping a bomb would most definitely be a negative event whereas in a stage where both of those are present, dropping a bomb can be seen as something positive. Furthermore, events such as MOVED_UP and WAITED lose relevance in later stages of training as we already expect our model to be comfortable at doing these effectively at that point and the extra rewards would just muffle the final reward sum. As such, in each step of our training process we would also adjust our reward values for efficient training.

It was also important for us to introduce a level of randomness to our agent’s exploration as otherwise it would go into valid looping actions. This is where the epsilon value of our model would come in and 30% of the turns it would force our agent to take a random action and explore previously unwitnessed game states. One major pitfall our agent would fall into was not waiting for the lingering explosion to end before going back into the explosion area. As such, a key fine tuning we performed in the late stages of training was that instead of 30% of the time taking a random action, our agent would just carry out the WAIT action. The motivation behind this was to ensure that the model experiences the rewards of waiting for the explosion to end more often and hence realizes to not move into it.

We carried out DQN Learning at the end of each round. We created a replay buffer class that stored the past 1000 transitions and sampled from the replay buffer to train our model.

3.2.3 Conjunctional model tuning (Hammad)

The approach we ultimately decided to use as our submission was a conjunctional approach where we used the faster but less robust imitation learning technique to get our model to a reasonable state and then used the slower but more robust DQN learning on the same model to get to the final version of our agent.

As alluded to previously, the biggest hurdle we faced when configuring our conjunctional model was that, despite many efforts, for a long period whenever we would feed our reasonably strong imitation model to DQN learning, the agent would always get worse and get locked into an action loop of moving up and down or left and right. We spent a lot of time configuring our reward structure as well as our training code, suspecting that one of these could be the issue with this behaviour. In the end, we finally discovered that the cause of this was the learning rate. Having realised that the two learning techniques react differently to the same learning rate, we reduced the learning rate for DQN learning substantially, and surely enough, we saw a dramatic improvement in our reward curve.

4. Experiments and Results

In this section we will go over the final agent models that we ended our project with as well as the nuances of each.

4.1 Least_Ultimate_Agent (Shivangi + Hammad)

The Least_Ultimate_Agent is our pure imitation learned agent. The agent was trained solely using the rule-based agent activity in the classic game scenario. The following image shows the loss curve of the agent throughout training.

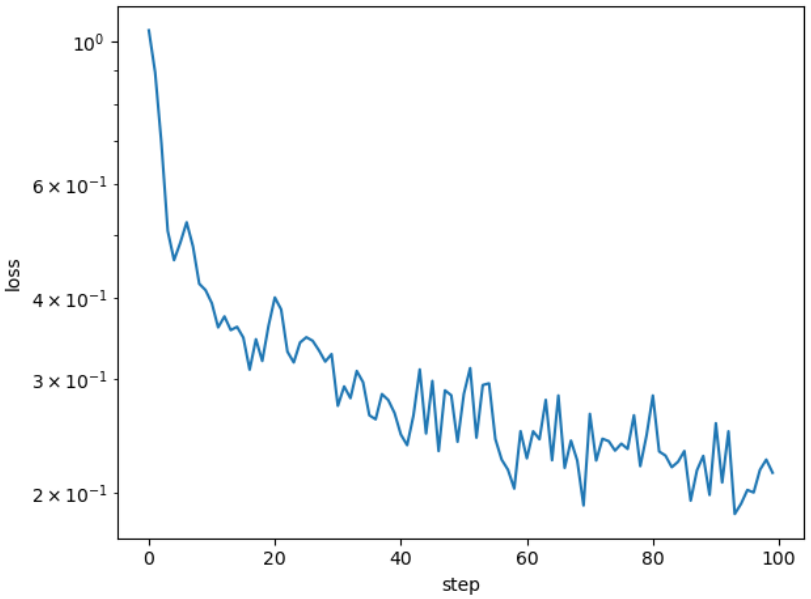


Image 4: Loss curve

The agent performance was quite impressive for the simple configurations it was trained on, but there were still problems that made it far from perfect. The agent was very good at collecting coins, and very rarely would it place bombs when it was not warranted. However, the major weakness of this agent was consistently its tendency to walk into explosions despite embedding very strong signals in the feature vector to inform it of danger. Nonetheless, this agent provided a very strong foundation for us to build upon and fine-tune using DQN Learning.

4.2 Less_Ultimate_Agent (Hammad)

For the second agent, we utilised the previously trained imitation model and carried out conjunctional tuning using the settings explained in the DQN model and training section. With DQN learning, we also kept track of the average reward collected by the agent as the replay buffer progressed. Image 6 shows the loss curve for the DQN training iterations after the imitation learning, and Image 5 shows the reward plots.

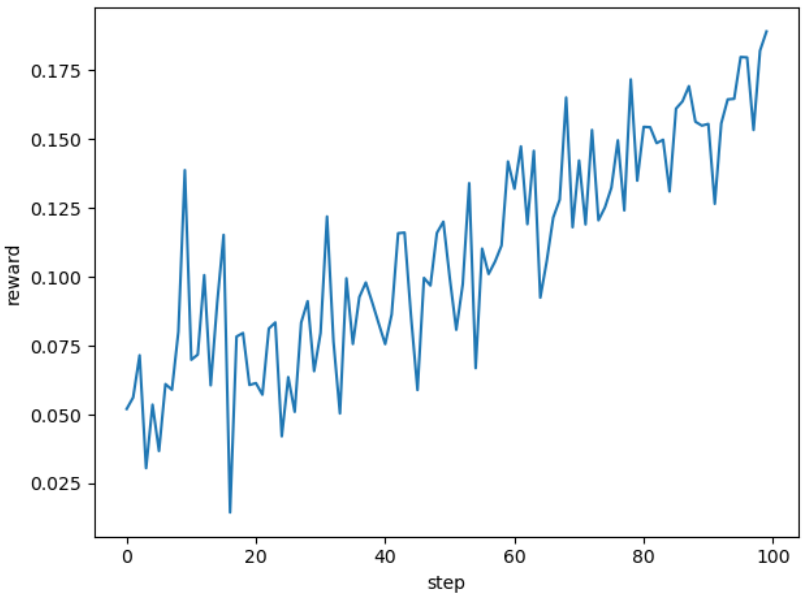


Image 5: Reward plot

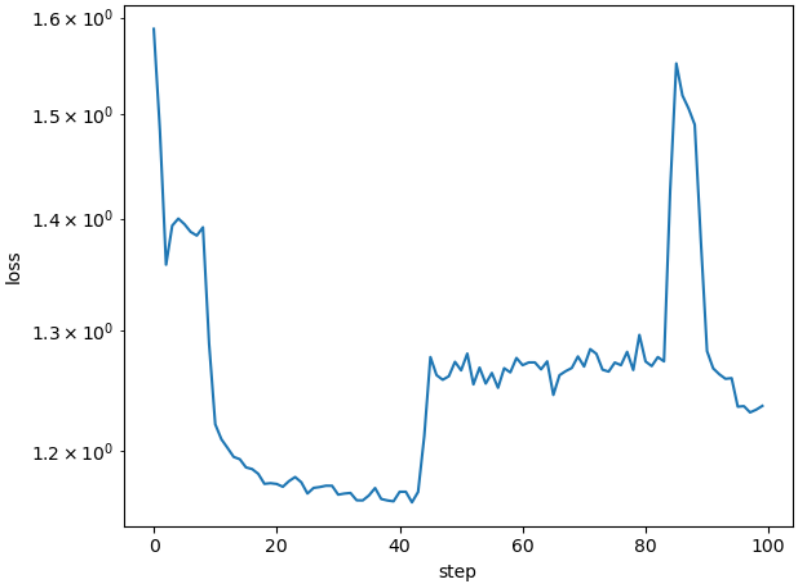


Image 6: Loss curve

As we can see, the loss curve for this agent is particularly peculiar. The reason for this is the use of scenario adjustment in our training. Up until 40% of our total training steps, we were training the model in a game arena containing only coins. In a game arena with only coins, there is no need to place bombs. As such, we immediately exclude one output that can be a source of loss, resulting in an overall very low loss value by the time the model is able to collect coins. At the 40% mark, we shifted to the game state containing crates. This setting presented the need to place bombs, and as the agent was killing

itself within the first few turns of the game, there were proportionally a lot more incorrect actions taken than correct ones, hence a sharp increase in loss. After 80% of our training steps, we started including enemies as well as empty stages, as mentioned in the training section. Here again, we can see a sharp increase in loss. This is because at this stage, the agent does not realise that it is supposed to eliminate enemy agents and, as such, does not try to avoid or attack them while getting attacked itself. Surely enough, the agent soon learns of this adjustment, and the loss begins to drop again.

Upon completing training, our agent had become decently capable at playing the game, yet there were still some pitfalls. The biggest issue with the agent was that it was once again not waiting as long as it should before going back into the lingering explosion. Ironically enough, the version of the model that had been trained up until the last stage was performing better in this regard.

4.3 Ultimate_Agent (Hammad)

This is the final model we made and submitted as our best model. Ultimate Agent builds upon the property we observed in the previous model that the agent has a tendency to not carry out the “WAIT” action. For this agent, we took two models: the Less_Ultimate_Agent model and the version of Less_Ultimate_Agent that was only trained up until the only crate game scenario. The reason for this is that the model trained only up to the crate game scenario shows a superior ability to avoid explosions. Once we had these two models, we utilised ensemble inference methods to help stabilise our fully trained model by taking a weighted sum of the two models. We attached a weight of 1 to the fully trained model and a weight of 0.2 to the partially trained model, and we took the argmax of the sum to decide our action. The intention for using this method was to help assist our final model in making a decision when it is not sure of the right action. The result of this experiment was a very impressive stabilisation in model performance. It would no longer move erratically and consistently avoid explosions with greater ability. We calculated the average loss and reward generated by this method on a replay buffer of 1000 entries and were able to reach an average reward of 0.2 and an average loss of 1.1.

5. Conclusion

In this report, we discussed the development and training of an agent to play the Bomberman game using Machine Learning techniques. We explored various approaches to feature shaping and model building, including imitation learning and Deep Q-Network (DQN) learning, to train the agent. Additionally, we discussed the design choices and feature engineering processes that were employed during the development of the agent.

The results of the training process led to the creation of three agent models:

- **Least_Ultimate_Agent:** This agent was trained solely using imitation learning, mimicking the rule-based agent's behaviour in a simple game scenario. While it performed well in coin collection, it struggled with avoiding explosions.

- **Less_Ultimate_Agent:** This agent utilised the previously trained imitation model as a foundation and underwent conjunctional tuning with DQN learning. It showed improvements in handling more complex game scenarios and learning to drop bombs strategically. However, it still had some issues with explosion avoidance.

- **Ultimate_Agent:** The final model, Ultimate_Agent, combined the strengths of the Less_Ultimate_Agent with another version of itself trained in a simplified game scenario. Ensemble inference techniques were used to stabilise its decision-making process. This model demonstrated a significant improvement in stability, especially in terms of avoiding explosions.

Our training process ensured that the agent started with a suitable and safe policy, reducing the risk of exploring dangerous actions during the early training phases. It also speeds up the learning process by leveraging existing rule-based expertise, resulting in a significant time reduction for training.

5.1 Outlook for Improvement (Hammad)

If more time were available to further improve the agent, the following steps could be considered:

- **Fine-Tuning:** Continue fine-tuning the model parameters, including reward values and learning rates, to further enhance the agent's performance in various scenarios.

- **More epochs:** With a large portion of our project time devoted to discovery and development, we definitely feel we could have allowed more time for our agent to train and improve its performance.

- **Dynamic Feature Extraction:** Develop a feature extraction method that adapts dynamically to the game state, which may help the agent better understand its environment and make more informed decisions. Particularly, we want to edit the field of view of the agent, as when there is no point of interest for the agent in its field of view, it often gets trapped in an action loop. At that point, we feel like expanding the agent's view of the arena will help it understand better what step to take.

- **Feature vector optimisation:** We also realised that we can combine our crate and enemy channels from our feature vector into a singular point of interest channel where the value signifies the amount of interest in the target. By doing this, in our opinion, we would be greatly optimising our feature shape by taking out a whole channel and ensuring faster learning.

- **Exclusion of walls information:** We hypothesised that since walls are a static feature in the game environment and have been left unchanged throughout the training process, removal of the walls channel in the feature matrix might have made for a more effective input feature. Without that information, the agent itself would have learned where it could and couldn't move.

By addressing these aspects, the agent's performance in the Bomberman game could be significantly enhanced, making it more capable of competing with skilled human players and tackling complex game scenarios.

6. References

- Book "Reinforcement Learning: An Introduction" by Richard S. Sutton and Andrew G. Barto. URL: http://incompleteideas.net/book/the-book-2nd.html

- YouTube series “Reinforcement Learning” by deeplizard. URL: https://www.youtube.com/watch?v=nyjbcRQ-uQ8&t=47s

- “Human-level control through deep reinforcement learning” - The original DQN paper by Volodymyr Mnih et al. URL: https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf

- DQN (Deep Q Networks) Implementation. URL: https://github.com/yenchenlin/DeepLearningFlappyBird

- Torabi, F. (2018, May 4), “Behavioral Cloning from Observation”, arXiv.org. URL: https://arxiv.org/abs/1805.01954v2

- Michael A. Nielsen, “Neural Networks and Deep Learning.” en. Determination Press, 2015. URL: http://neuralnetworksanddeeplearning.com

- P. M. Kebria, A. Khosravi, S. M. Salaken and S. Nahavandi, "Deep imitation learning for autonomous vehicles based on convolutional neural networks," in IEEE/CAA Journal of Automatica Sinica, vol. 7, no. 1, pp. 82-95, January 2020, doi: 10.1109/JAS.2019.1911825. URL: https://ieeexplore.ieee.org/abstract/document/8945486