Machine Learning Essentials
Reinforcement Learning for Bomberman

Anusha Chattopadhyay, 3767958
Masters Student- Data and Computer Science
Link to GitHub Repository
Team Name - Dodge

September 30 2024

Abstract

Anusha

This document serves as the final project report for the subject Machine Learning Essentials, focusing on the application of Reinforcement Learning in the game Bomberman. The primary objective of this project was to develop an intelligent agent capable of playing Bomberman autonomously. The following report shall document the problem statement, the model structure and the theory behind it. It shall also document the the various attempts at improving the model. This project uses Deep Q Learning to attempt to teach the agent model. After turning the game state into an image. The CNN model processes the visual input to make informed decisions within the game environment. To further aid the learning process, specific rules and guidelines were implemented, ensuring the agent could effectively learn and adapt to different scenarios within the game.

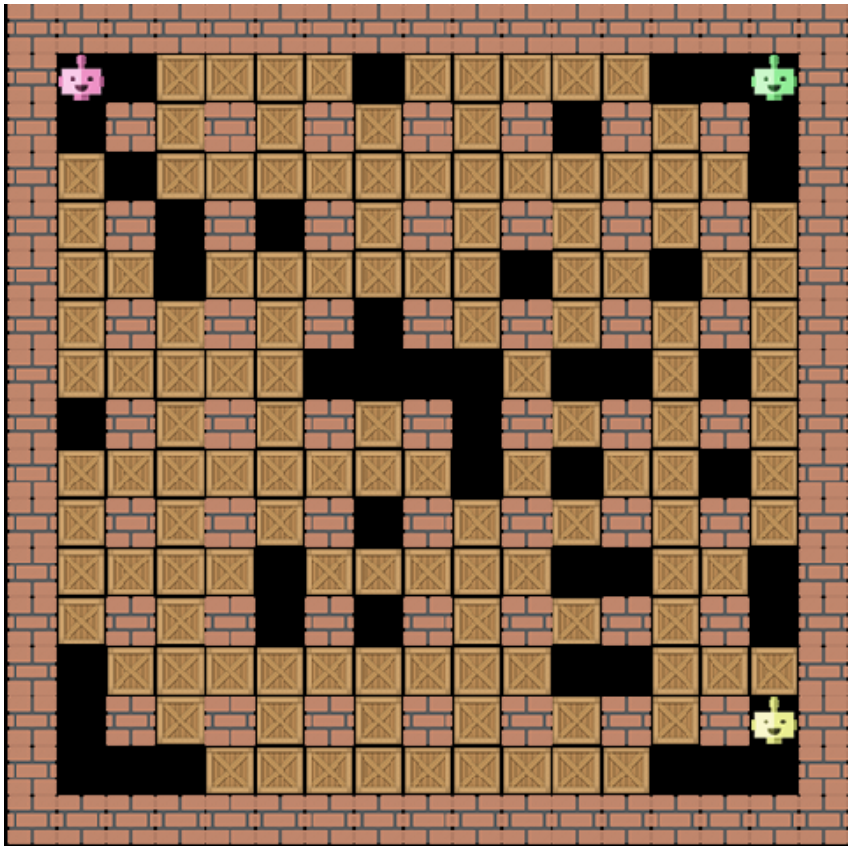

Figure 1: Example of the game starting with 3 Agents

Contents

1 Introduction **4**
- 1.1 Rules and Game Information . . . 4
  - 1.1.1 Game Arena . . . 4
  - 1.1.2 Agent . . . 4
  - 1.1.3 Items . . . 4
  - 1.1.4 Actions . . . 4
  - 1.1.5 Win Condition . . . 5

2 Challenges to face from the beginning **5**
- 2.0.1 Input size . . . 5
- 2.0.2 Walls and Bombs . . . 5
- 2.0.3 Delayed Rewards . . . 5

3 Background **6**

4 Project Planning **6**

5 Methods **7**
- 5.1 Q-Learning . . . 7
- 5.2 Deep Q-Learning . . . 7
- 5.3 Convolutional Neural Networks (CNNs) . . . 8

6 Technical Specifications **8**
- 6.1 Code Structure . . . 8

7 Implementation **8**
- 7.1 Final Project Agent Code File Structure . . . 9
- 7.2 Model . . . 10
  - 7.2.1 Model Input . . . 10
  - 7.2.2 Model Structure . . . 11
  - 7.2.3 Training . . . 11
  - 7.2.4 Reward System . . . 12

8 Improvements **12**
- 8.0.1 Rules Implementation . . . 13

9 Experiments and Results **13**
- 9.1 Setup . . . 13

10 Conclusion **14**

11 Future Work **16**

1 Introduction

Anusha

In the final project for Machine Learning Essentials was to use reinforcement learning techniques to train an agent to play the classic arcade game Bomberman.

1.1 Rules and Game Information

In the following section the key rules of the game are explained. This is provide context for the further actions for the attempted solution for the problem.

1.1.1 Game Arena

The Game Arena is a dynamic environment that includes various elements such as Agents, Stones, Coins, Bombs, and Crates. The boundary of this arena, often referred to as the game state, is defined by a Wall constructed from Stone. The game progresses over a predetermined number of rounds. While each round retains the cumulative points from previous rounds, the physical layout of the Arena is reset at the beginning of each new round. Figure 1 provides a visual representation of the arena setup.

1.1.2 Agent

An Agent is a "Player" in the game. The main task is to design an Agent that can play and hopefully win a game with Reinforcement Learning.

1.1.3 Items

- Stone - These are permanent 'walls' within the game arena. Agents cannot move beyond or through a Stone. Stones are indestructible and cannot be destroyed by bombs.
- Crate - Crates serve as temporary 'walls' in the game arena. Similar to Stones, Agents cannot pass through Crates. However, unlike Stones, Crates can be destroyed using Bombs, which clears the path. Destroying a Crate may reveal a hidden Coin.
- Bomb - Bombs are explosive devices that detonate shortly after being placed. The explosion affects the four adjacent directions around the Bomb. Any Crates in the explosion's path will be destroyed, and any Agents caught in the explosion will be eliminated.
- Coin - Coins are valuable items that appear in the arena. Initially, they are hidden behind Crates. Once a Crate is destroyed, the Coin can be picked up by an Agent.

1.1.4 Actions

There are the following actions the Agent can perform and the possible result of these actions.

- Move: Agents have the ability to move in four primary directions: Left, Right, Up, and Down. Additionally, they can choose to Wait, remaining in their current position. Attempting to move beyond a wall or through a crate is considered an invalid action and will not be executed.
- Place Bomb: Agents have the option to place a Bomb at their current location. The strategic placement of a Bomb can result in the elimination of rival agents, which rewards the agent with points. However, agents must be cautious, as they can also be killed by their own Bomb if they remain in the explosion's path.
- Coin pickup: When an agent reaches a Coin, they automatically pick it up, thereby gaining points. Coins are valuable items that contribute to the agent's overall score.

1.1.5 Win Condition

The ultimate goal of the game is to accumulate the highest number of points by the end of all the rounds. The agent who successfully gathers the most points throughout the entire duration of the game emerges as the winner. This means that strategic moves, effective use of bombs, and timely coin pickups are crucial for securing victory. The agent with the highest score at the conclusion of all rounds is declared the champion.

2 Challenges to face from the beginning

Anusha

2.0.1 Input size

One of the initial challenges was understanding how to effectively process the game state information. The game state could potentially encompass a large arena, resulting in a substantial amount of input data. This required developing strategies to manage and interpret this extensive information efficiently.

2.0.2 Walls and Bombs

Another significant challenge was determining how to reduce the likelihood of the agent making fatal mistakes, such as killing itself or attempting invalid moves like walking through walls. Mechanisms needed to be implemented to ensure the agent could recognize and avoid these hazards, thereby improving its overall performance and decision-making capabilities.

2.0.3 Delayed Rewards

The agent also faced difficulties in learning from its mistakes due to the delayed nature of rewards. This delay, combined with the dynamic nature of the game state, made it challenging to find timely solutions. The agent required a considerable amount of time to understand the consequences of its actions and adjust its behavior accordingly, which added complexity to the training process.

3 Background

Anusha

For this project the approach tried has been Deep Q Learning.

The main sources behind building the solution has been:

"Playing Atari with Deep Reinforcement Learning" : Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller [2]
and,
The resulting official Pytorch.org tutorial: "Reinforcement Learning (DQN) Tutorial": Adam Paszke, Mark Towers[3]

4 Project Planning

Anusha

Due to unforeseen circumstances, the project ended up being completed by a single member. Consequently, this report details the solution developed by the sole remaining agent. The team name, "Dodge" was inspired by the agent's favorite internet-famous animal, known for its iconic look of confusion.

However the project was broken into these following stages of development:

1. Tackling the "Possibly Large Input" Problem:
    - The initial challenge was to address the issue of handling potentially large inputs. The agent aimed to start with a simplified model to ensure a clear understanding of the problem space.
    - This involved researching various techniques for input management and selecting the most feasible approach for the project's scope.
2. Model Selection and Initial Implementation:
    - The next step was to choose an appropriate model and create a basic implementation that functions without errors.
    - At this stage, the primary goal was to develop a working prototype, regardless of its performance.
    - This involved setting up the development environment, selecting libraries and frameworks, and writing the initial code to ensure the model could process inputs and produce outputs.
3. Improving the Initial Model with Rules:
    - To enhance the model's performance, the agent sought alternatives to the initial random approach used during the early stages of model training.
    - This involved incorporating rules and heuristics to guide the model's learning process.

- The agent experimented with different rule-based techniques to improve the model's accuracy and reliability.

4. Training the Model:

   - The training phase began with the model being trained on a CPU using an Intel i5 processor.
   - This initial training helped identify any potential issues and allowed for iterative improvements.
   - If time permitted, the agent planned to conduct an extended training session on Google Colab using the `--no-gpu` tag.
   - This would leverage the cloud-based platform's resources to further refine the model and enhance its performance.

5 Methods

5.1 Q-Learning

Q-Learning is a type of reinforcement learning algorithm that helps an agent make optimal decisions in an environment to maximize cumulative rewards.
The key aspects of Q-learning can be broken down into:

- Action and Environment: The agent takes an action, and the environment reacts in a certain way after the action. The environment's state changes based on the action taken by the agent, and this new state is observed by the agent.

- Reward Allocation: The results of the change in the environment can be used to reward the agent or assign points to the Q-table. For example, a good result/action is a positive allocation of points (such as defeating a rival agent), and a bad result/action is a negative allocation of points (such as killing your own agent with your own bomb). The reward signal helps the agent learn which actions are beneficial and which are detrimental.

- Q-table: A lookup table where each cell represents the expected future rewards for taking a specific action in a given state. The table helps the agent decide the best action to take in each state to maximize cumulative rewards. The Q-values are updated iteratively using the Bellman equation:

  $$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

  where α is the learning rate, r is the reward, γ is the discount factor, and $\max_{a'} Q(s',a')$ is the maximum expected future reward for the next state s'.

5.2 Deep Q-Learning

This is Q-learning, but a neural-network is used.
Deep Q-Learning extends Q-learning by using a neural network to approximate the Q-values instead of a lookup table.The neural network takes the state as input and outputs Q-values for all possible actions.

Key components of Deep Q-Learning include The Experience replay. This technique involves storing the agent's experiences (state, action, reward, next state) in a replay buffer. During training, random samples from the buffer are used to break the correlation between consecutive experiences. This improves the learning stability.

5.3 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are a class of deep neural networks that are particularly effective for processing grid-like data, such as images.
Key features of CNNs include:

- Convolutional Layers: These layers scan the input data in small sections, allowing the network to learn spatial hierarchies of features. Convolutional layers apply filters to the input data to detect patterns such as edges, textures, and shapes.
- Parameter Sharing: By using the same filter across different parts of the input, CNNs significantly reduce the number of parameters, making the network more efficient.
- Pooling Layers: Pooling layers reduce the spatial dimensions of the input. Common pooling operations include max pooling and average pooling.
- Fully Connected Layers: After several convolutional and pooling layers, the high-level features are flattened and passed through fully connected layers.

6 Technical Specifications

Anusha

The solution has been written in pytorch. The UI of the framework of the game used pygame. Thus when training on colab, –no-gui tag was necessary.
Training was done on cpu on an Intel Core i5

6.1 Code Structure

The Agent was created in a folder in the Agent folder of the repository. This used the base template of the tpl agent. This folder consisted of few other agents to help as reference while understanding how the repository code works. These could also be used to play against Dodge's Agent.
These agents were the fail-agent: destined to fail, peaceful-agent: committed not to bomb, random-agent: chooses an action randomly and rule-based-agent: chooses the action based on a comprehensive logic system.

7 Implementation

This section provides an overview of the various details involved in the implementation of the Agent.

7.1 Final Project Agent Code File Structure

The Agent code consisted of the following files with the key functions in each file.

1. callbacks.py

   - def act(self, game-state: dict):
     This is the function that dictates what the agent does and thus exports the final action.
     The action can be ['UP', 'RIGHT', 'DOWN', 'LEFT', 'WAIT', 'BOMB'].
   - def setup(self):
     This is called once when loading each agent.

2. data.py:
   This file was created to be the dataloader needed for the model

   - def create-input(game-state: dict):
     Converts the the game-state into an image. Details provided later on in the Model Input section in 7.2.1.

3. dqn.py:
   This contains the model

   - class DQN(nn.Module):
     The Model used for learning is defined in this file. Details in the Model Structure section in 7.2.2.

4. train.py:
   The following key variables are there in this file:

   BATCH-SIZE = 128
   GAMMA = 0.99
   EPS-START = 0.9
   EPS-END = 0.1
   EPS-DECAY = 0.999
   TAU = 0.001
   LR = 1e-4

   - def setup-training(self):
     Initialise self for training purpose.
   - def game-events-occurred(self, old-game-state: dict, self-action: str, new-game-state: dict, events: List[str]):
     Called once per step to allow intermediate rewards based on game events.
   - def train(self):
     The training function.
   - def end-of-round(self, last-game-state: dict, last-action: str, events: List[str]):
     Called at the end of each game or when the agent died to hand out final rewards.

- reward-from-events(self, events: List[str]):
  Rewards for each event is set. Further details in the Reward System section in 7.2.4
- store-experience(self, state, action, reward, next-state, done):

- def update-target-model(self):

7.2 Model

This section talks about key information about the model and how it was structured.

7.2.1 Model Input

The game state was turned into an image by adding different intensities to different items in the game state and then used as an input for the CNN.
The image as in input tensor in the form of a heat map is visualized in Figure 2

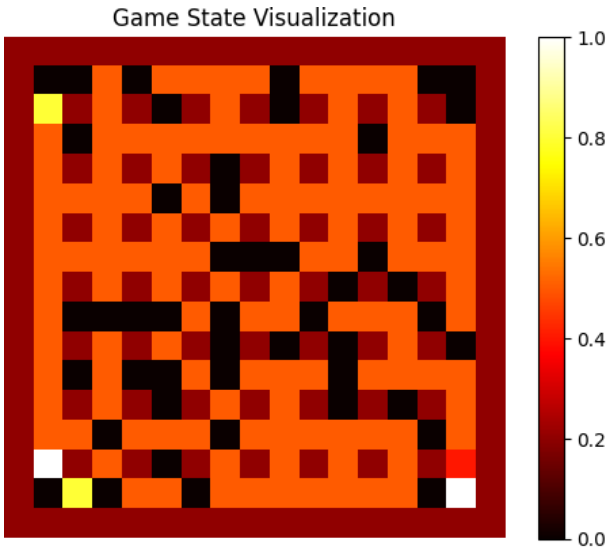

Figure 2: Heatmap created of game-state

Each item was assigned an intensity value. For example, the agent was given a value of 0.2.

The intensities for each item were decided arbitrarily, with the only stipulation being that bombs and explosions should have the same intensity, as the agent should avoid them regardless.
There was a consideration to add an additional meaning to the heat map, aiming to make the board "cooler" or "hotter"; however, this possibility was not explored.

Possible issues with this approach include the use of pooling layers in Convolutional Neural Networks (CNNs). Pooling layers reduce the spatial dimensions (width and height) of the input feature maps, which helps in reducing the computational load and controlling overfitting. However, this process can sometimes make it harder for a CNN to effectively read greyscale images. This issue can also apply to the heatmap approach.

7.2.2 Model Structure

- Components
  - Convolutional Layers:
    - `conv1`: A convolutional layer with 1 input channel, 32 output channels, a kernel size of 4, stride of 2, and padding of 1.
    - `conv2`: A convolutional layer with 32 input channels, 64 output channels, a kernel size of 4, stride of 2, and padding of 1.
    - `conv3`: A convolutional layer with 64 input channels, 64 output channels, a kernel size of 3, stride of 1, and padding of 1.
  - Adaptive Pooling:
    - `adaptive_pool`: An adaptive average pooling layer that outputs a fixed size of (11, 11).
  - Fully Connected Layers:
    - `fc1`: A fully connected layer that takes the flattened output from the convolutional layers and adaptive pooling, with 512 output units.
    - `fc2`: The final fully connected layer that outputs the Q-values for each action, with the number of actions specified by `n_actions`.
- Forward Pass
  1. Convolutional Layers: The input x is passed through the convolutional layers (`conv1`, `conv2`, and `conv3`), each followed by a ReLU activation function to introduce non-linearity.
  2. Adaptive Pooling: The output from the last convolutional layer is passed through the adaptive pooling layer to ensure a fixed output size of (11, 11).
  3. Flattening: The pooled output is flattened into a 1D tensor to be fed into the fully connected layers.
  4. Fully Connected Layers: The flattened tensor is passed through `fc1` with a ReLU activation function. Finally, the output is passed through `fc2` to produce the Q-values for each action.

7.2.3 Training

Training is conducted through a series of rounds, with rewards being distributed at the conclusion of each round. In the initial phase of training, actions are determined by preset logic. For instance, a word might be chosen at random, with each potential move having a predefined probability of being selected. This phase continues for a certain duration and under specific conditions. Once these conditions are met, the model transitions to predicting moves based on the knowledge it has accumulated up to that point. The intricacies of this process are elaborated upon in the sections below.

To decide when to use preset logic or the model to predict, three main variables are involved.

- epsilon = EPS.START

- epsilon-min = EPS.END
- epsilon-decay = EPS.DECAY

The logic loop works as follows

- if self.train and random.random() less-than self.epsilon
  Do action according to predefined logic/random
- else
  Predict according to the Q-values of the model
- while training:
  if self.epsilon greater-than EPS-END
  self.epsilon *= EPS-DECAY

7.2.4 Reward System

End of each round, Rewards would be allocated. The rewards had the following rules:

- e.COIN-COLLECTED: +100,
- e.CRATE-DESTROYED: +50,
- e.COIN-FOUND: +100,
- e.KILLED-OPPONENT: +100,
- e.KILLED-SELF: -1000,
- e.SURVIVED-ROUND: +10,
- e.INVALID-ACTION: -10,
- e.BOMB-DROPPED: +0,
- e.WAITED: -1

Killed-Self was the action that was most discouraged at -1000 points and BOMB was left at 0. WAIT was discouraged at -1 as the model keep on moving means more possible moves.An additional reason is that WAIT was kept as a default option for the rule based training, thus the discouragement was added in hope the model does not learn to just WAIT.

8 Improvements

The subsequent section provides an in-depth discussion regarding various methods and approaches that were employed to enhance and refine the Agent. In attempt to make the Agent perform better.

8.0.1 Rules Implementation

The initial implementation involved the model selecting an action at random, guided by a set of predefined probabilities. To improve upon this, three different rule-based approaches were explored:

- Run to safety the next turn
  Make the model consider bombing only if it can move to safety immediately This approach aimed to prevent the model from self-destructing. The idea was to have the model make decisions based on its proximity to the bomb or the explosion zone. The bomb would only be placed if the agent could ensure it could move to a safe location immediately after placing the bomb. However, this method was not very successful. Although the agent could find a safe spot, it often failed to move away in time, resulting in its destruction.

- Mimic the expert's behavior
  In this approach, the agent adopted strategies from a rule-based expert agent. This method showed more promise during training. As it performed more diverse moves and killed itself less often. However, during testing, the agent frequently got stuck in a repetitive "ONE MOVE" decision loop. This issue was also observed in the previous random rule approach, but it was more pronounced in this method, occurring almost every time the agent was trained afresh for 300 rounds.

- Incorporate a hybrid approach
  This method combined elements from both the random and expert-based approaches. The agent would initially follow the expert's rules but switch to a probabilistic model when certain conditions were met. This hybrid approach aimed to balance the rigidity of rule-based decisions with the flexibility of probabilistic choices. While this method showed some improvement, it still faced challenges in dynamic environments where quick adaptation was necessary.

Each of these approaches aimed to enhance the decision-making capabilities of the agent, but they also highlighted the complexities involved in creating a robust rule-based system. Future work could involve refining these rules or integrating them with more advanced machine learning techniques to achieve better performance.

9 Experiments and Results

Anusha

For the experiments we decided to test 3 versions of the agents. Where the logic based section of the agent is different.
These Agents are

9.1 Setup

- Only-random:
  This agent determines its actions purely based on randomness. Specifically, it decides its next move using the following line of code:

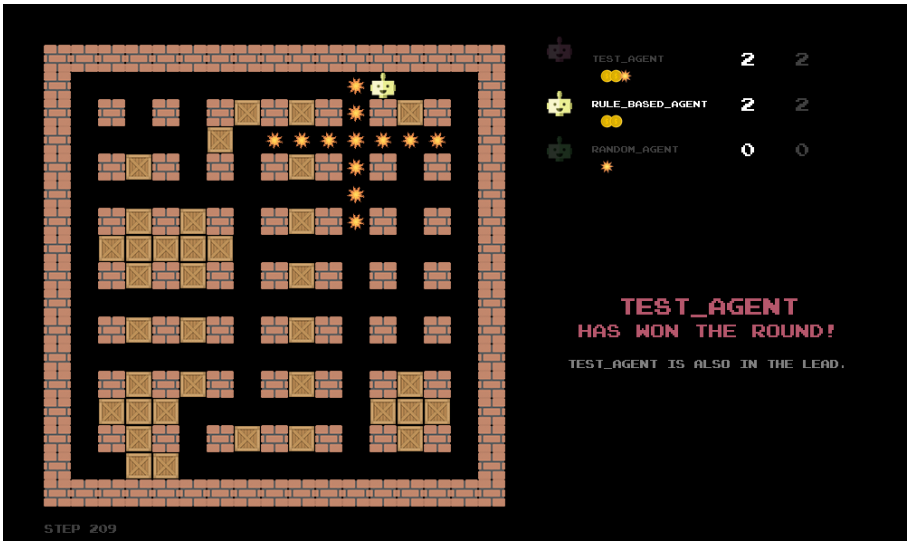

Figure 3: A rare event when the Agent won, however it did not survive the round

```
action = np.random.choice(ACTIONS, p=[.21, .21, .21, .21, .05, .11])
```

This means that each action is chosen with a probability of 21%, except for one action with a 5% probability and another with an 11% probability.

- Bomb-avoid:
  The agent employs a strategy to avoid danger from bombs. Before placing a bomb, it checks whether its next move will lead it into a section where the bomb's explosion could harm it. Only if the agent determines that it will be safe from the bomb's blast radius will it proceed to place the bomb.

- Rule-based-agent-mimic:
  This agent mimics the logic of a rule-based agent. It follows the same decision-making process as the rule-based agent when it is not relying on a model to decide its actions. Essentially, it uses predefined rules to determine its behavior in various situations.

- Randomly-rule-based-agent-mimic:
  This agent also mimics the rule-based agent, but with an added layer of randomness. It randomly decides whether to follow the rule-based logic or to choose an action at random. There is a 1 in 5 chance (20% probability) that it will select an action randomly instead of using the rule-based logic.

These Agents were tested against 3 other rule-based-agents. The results are in Table 1. This documents the number of times the agents did certain actions. As per the logged events.

10 Conclusion

Anusha

This concludes that the rules did not help substantially when it came to surviving. The

Metric	Only-random	Bomb-avoid	Rule-mimic
Died	863	991	975
Bombs	3490	3980	7536

Table 1: Tabular Section of Rounds out of 1000 rounds

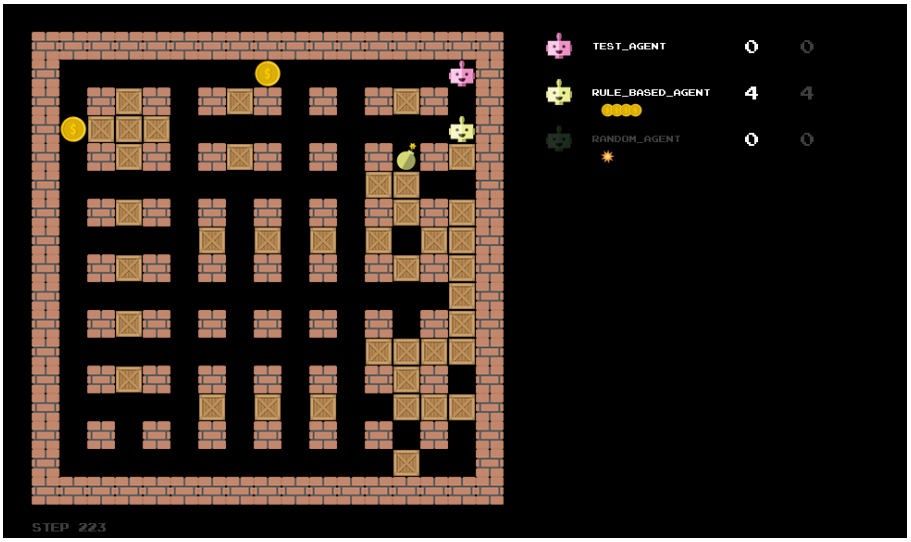

Figure 4: Instance where Q-Value overestimation may be happening, the model stays in the same position throughout the testing phase

Rule-mimic was the most violent. Interestingly, the bomb-avoidant one did not live up to it's name and was more violent than random. This could be because the probability of bombing was very low in the random at 0.5. However The rule based systems had fewer Invalid actions.
These results indicate that the improvements required should prioritize and focus on the model itself.
The reasons for the model not working well could be the following:

- Pooling Layers in CNNs: Pooling layers are used to reduce the spatial dimensions (width and height) of the input feature maps. This reduction helps in decreasing the computational load and controlling overfitting. However, this process can sometimes make it challenging for a CNN to effectively interpret greyscale images. This difficulty can also extend to the heatmap approach, where important spatial information might be lost during pooling.

- Instability in Training: The model updates based on consecutive experiences, which are often highly correlated. This correlation can lead to instability and poor convergence, as the model might be biased to recent experiences. The lack of diverse training data can cause the model to become biased towards specific patterns, reducing its overall effectiveness.

- Q-Value Overestimation: The model might suffer from Q-value overestimation, where it overestimates the value of certain actions due to the lack of diverse experiences. The model might prioritize actions that are not truly beneficial in the long run.

- Poor Generalization: The model might not generalize well to different states and actions, as it is trained on a narrow set of recent experiences. This narrow training scope can limit the model's ability to adapt to new and varied situations, reducing its overall performance and reliability.

To address these issues, it may be beneficial to explore the replay functions of the implementation. Replay functions can help by storing a diverse set of past experiences and using them to train the model. This approach can mitigate the problems of instability, Q-value overestimation, and poor generalization by providing a broader and more varied training dataset.

11 Future Work

Anusha

Given more time, the first task would be to address some of the issues stated in the conclusion.
If continuing with an image-input based approach, it will be crucial to explore different ways to express the game-state as an image.
This could involve experimenting with various image representations and preprocessing techniques to enhance the model's ability to interpret the game environment. For instance, different color schemes, resolutions, and segmentation methods could be tested.
Additionally, exploring alternative methods for processing game data, such as using recurrent neural networks (RNNs) or transformers, could potentially improve the model's ability to interpret and respond to dynamic environments.
RNNs, with their ability to handle sequential data, could be particularly useful for capturing temporal dependencies in the game state. Transformers, known for their powerful attention mechanisms, could provide a better understanding of the game environment by focusing on the most relevant parts of the input data.
Implementing advanced techniques like prioritized experience replay could mitigate the instability in training by ensuring a more diverse set of experiences for model updates.
This technique prioritizes more informative experiences, which can lead to faster and more stable learning.
To combat Q-value overestimation, employing Double Q-learning[1] or other variants could provide more accurate value estimations.
Double Q-learning reduces the bias associated with overestimation by decoupling the action selection from the Q-value update, leading to more reliable learning outcomes.
Additionally, exploring other variants such as Dueling Q-networks[4], which separate the estimation of state values and advantages for actions, could further increase the success of the learning process.
Outside the realm of machine learning, various strategies related to the game BOMBERMAN itself could be examined and implemented to improve overall performance. This could include refining the rules and mechanics of the game to make it more effective for learning, as well as incorporating domain-specific knowledge to guide the agent's behavior.
By addressing these areas, the overall performance of the system could be significantly enhanced, hopefully resulting in a better performing model.

References

[1] Hado van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-learning". In: *CoRR* abs/1509.06461 (2015). arXiv: 1509.06461. URL: http://arxiv.org/abs/1509.06461.

[2] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG]. URL: https://arxiv.org/abs/1312.5602.

[3] Reinforcement Learning (DQN) Tutorial. *Adam Paszke,Mark Towers*. URL: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html.

[4] Ziyu Wang, Nando de Freitas, and Marc Lanctot. "Dueling Network Architectures for Deep Reinforcement Learning". In: *CoRR* abs/1511.06581 (2015). arXiv: 1511.06581. URL: http://arxiv.org/abs/1511.06581.