Intro to Artificial Intelligence
CS4341

Project 2: Bomberman

Submission by:
Nicholas Delli Carpini
Justin Cheng
Josue Contreras

Abstract

The goal of this project was to build an AI agent using reinforcement learning to play the classic Bomberman game. The agent was capable of beating each variant in the two scenarios given. This agent was also able to blow up walls, monsters, and and tried to get to the goal quickly to accomplish a better score.

Introduction

Bomberman is a classic strategy maze game that consists of reaching the exit with the highest possible score. There are various implementations of this game. The one used for this project was implemented by Professor Pinceroli and can be found in the Nest Labs github[1] repository. This implementation of Bomberman consists of two different scenarios. Each has five variants with increasing difficulties. The two scenarios have their own maze configuration, the variable that increases the difficulty is the types of monsters that are placed in each variant. There are three types of monsters: random, self-preserving, and aggressive. The aggressive monster is able to increase its aggressiveness by the cells it is allowed to detect the character. There also exists a third scenario that was not present in the code given, but it was mentioned that the classic game of Bomberman also allows for a scenario with more than one character. In this last scenario the goal of staying alive as no exit exists and to blow up the other characters before yourself.

Unlike other simple deterministic games like Connect-N and tic-tac-toe, Bomberman is a stochastic game where randomness exists and the next play can only be predicted to a certain extent. Therefore, decision trees can reach levels with exponential possibilities. To be able to predict the best next move agents are trained to learn through various new methods of reinforcement learning. This paper will focus on an implementation of Q-learning using reinforcement learning to train the agent in the game of Bomberman.

In Q-learning a set of rewards are given by the environment to the agent as it explores. After some exploration time it starts to determine and pick the best actions based on its state. This then allows the Q-learning agent to exploit and find the best solution. Even though the best Q values can obtain it will always have to learn more as environments change because of randomness. This is called regret, but in the current implementation of Q-learning in this paper because of time constraints it was not implemented.

Q-Learning Design

Q-learning is a type of model-free reinforcement learning algorithm. The decision to choose this particular algorithm was based on the need to build a learned function to solve different variants of different scenarios, and to satisfy the requirement of utilizing Q-learning for graduate level credit for this course. There are two world scenarios, each with 5 variants for a total of 10 different world variations of Bomberman. As such, integrating a learning algorithm such as Q-learning is a preferred strategic decision since in theory, a learned solution can be obtained to solve all of the variants in a consistent manner.

[1] https://github.com/NESTLab/CS4341-projects

Constructing the Q-learning algorithm requires creating, populating, and updating a Q-table. A Q-table is a table of referenced values for a given action in any given state. For our application, a Q-table was constructed to record the Q values for each action for any given state of the world in which the Bomberman plays.

To create the Q-table used in the Bomberman algorithm, the actions were first formalized. In any given cell, the Bomberman has the option to stay still (essentially passing a turn) or move to any of the eight surrounding cells. This gives a total of nine countable actions. The Bomberman also has the ability to place a single bomb in its current cell, which can be performed along with any of the nine previous actions, which gives a total of 18 possible actions at any given cell, which is not the same as legal actions, ie. moving into a cell out of bounds or occupied by a wall, and is accounted for in the computation of the Q value. Each row entry in the table is identified by a state, which comes from the world environment, thus giving N states for any of the 18 actions. Shown below in Figure 1 is the breakdown of the different actions and how they are read in the algorithm, as well as an example Q-table.

Actions		
Move X	Move Y	Place Bomb
-1	-1	0
-1	-1	1
-1	0	0
-1	0	1
-1	1	0
-1	1	1
0	-1	0
0	-1	1
0	0	0
0	0	1
0	1	0
0	1	1
1	-1	0
1	-1	1
1	0	0
1	0	1
1	1	0
1	1	1

Bomberman Q-Table				
	Action 1	Action 2	...	Action 18
State 1	0	0	0	0
State 2	0	0	0	0
...	0	0	0	0
State N	0	0	0	0

Figure 1: Bomberman Action and Q-Tables

For each possible movement (action) to the cells surrounding the bomberman's cell, the direction can be indicated as a change in the vertical and horizontal axis, x and y respectively. A -1 represents moving to the left or up, a 1 represents moving to the right or down, and a 0 represents no motion in the x or y axis. To capture bomb placement, 1 represents the action to place a bomb and 0 to not place a bomb (default).

The state of the Bomberman world or environment is dependent on the location of the main character, any other character that is on the board, any monsters, any walls, any bombs or explosion cells, and the location of the exit. This is captured by the observable field or the view. However, given the size of the game board and all the cells that can be occupied by characters, monsters, bombs, and explosions, there is an obscenely large number of combinations and thus a large number of states to be accounted for. The need to account for variations in wall

placement is necessary due to different world variations between the given scenarios. When constructing Q-tables the biggest challenge is to formalize the world as a state. For the first implementation of the agent the state of the Q-table saved the entire world as a state. However, this state proved to have too many variables that made learning from the state near impossible. There are issue with tracking all of these states, which primarily include length processing time of looking up and updating values on the Q-table even if it is assumed that all possible states have been recorded, as well as the constraint of file size with storing all of the states as a look-up table to be used for all 10 variants. To avoid having to store a Q-table with an infinitely large number of states in a large file, the view of the world was limited to be around the main character in the environment. Discretizing the limited view produces a state that captures only the character, monsters, walls, bombs, and explosion cells inside the view as opposed to the view of the entire game. Shown below in Figure 2 is a summarized process of simplifying state extraction from the given world.

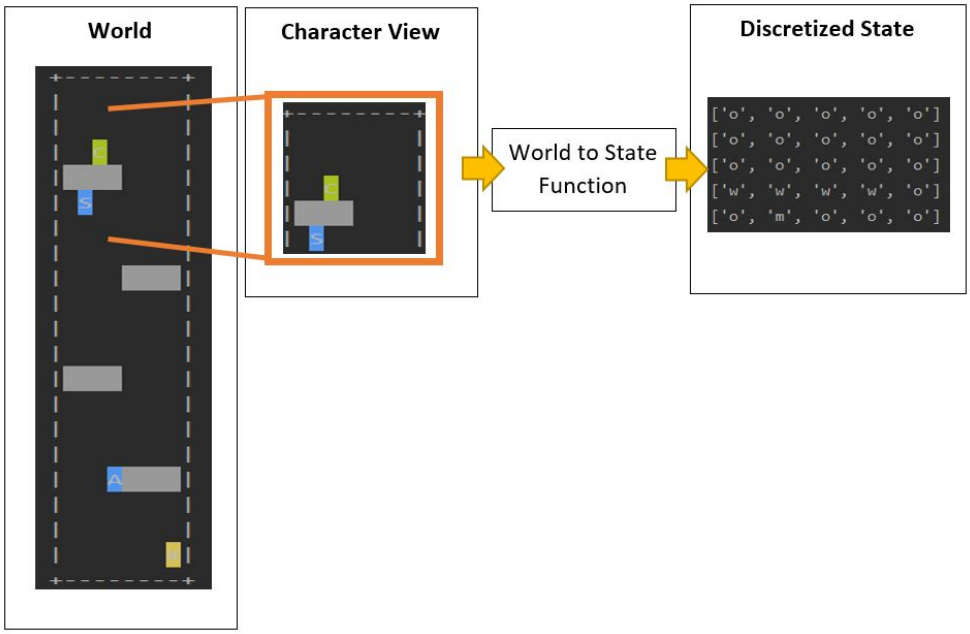

Figure 2: World to Simplified State Processes Diagram

As seen in the Figure, the overall view of the world is cropped to just the view around the character, which is passed into a function that discretizes the view into a state. A cell is marked for example with a ‘w’ if there is a wall, with ‘m’ if there is a monster, and with ‘o’ if the cell is otherwise unoccupied. Increasing the size of this view will allow for taking into account more possibles and thus tracing more states. This would be beneficial if computing time and storage did not factor into the overall concern for functional performance for the project.

Reward values were assigned based on events that would occur in the next state. For Bomberman, the reward of an action taken involves the presence of monsters, bombs, explosion blocks, walls, and other characters. For the purpose of creating the Q-table, arbitrary values were assigned to rewards depending on the action-state outcome associated with the

reward. The general idea for assigning rewards is to have lethally bad moves to have a large negative value and winning or good moves to have large positive values. The reason for this arrangement is to both encourage “good” plays and discourage “bad” plays. In our case, “good” plays would be defined as making progress towards the exit in the map and “bad” plays would be self-destructive actions, which include blowing oneself by walking into an explosion cell (appears after a placed bomb explodes) or walking into a monster (occupying the same cell results in death). At the basic level of survival and getting to the exit, we have defined the certain results for state-action pairs as having the following rewards for Reward r:

Initial reward (life penalty) **r = 5000 / -agent.score * 0.5**
Destroying a wall **r = +10**
Destroying a wall (in A-Star path) **r = +20**
Destroying a monster **r = +50**
Destroying a character **r = +100**
Following A* suggestion **r = +3**
Following A* ‘general’ suggestion **r = +1**
Not following A* **r = -1**
Running directly away from monster **r = +3**
Running ‘generally’ away from monster **r = +1**
Death (monster, bomb explosion) **r = -500**
Exit results in r = **+10000**

Within the game of Bomberman, there is a scoring mechanism. The score for a run of the game is dependent on which mode in which the game is being played. The variants all use the escape mode, in which scoring is depending on how fast the exit is reached given the max time limit. Survival in terms of how long the character lasts in the game is also a considered factor in that one point is given, but the longer the character stays in the game in this mode, the lower the end score if the character escapes. This is due to the fact that 2-times the remaining time is added to the overall score. There are also certain events based on actions performed that can add to the score:

The Q-learning implementation created for this project sufficiently solves scenario 1 variant 5, the highest variant in the first scenario, with the assistance of A* pathfinding. Preference for actions that adhere to the path returned by A* by adding to the reward added for that action. However, unlike all variants in scenario 1, variants in scenario 2 have walls that completely block off the exit. In this scenario, A* cannot return a path to the exit while considering the walls. However, A* can still generate a path if the walls were theoretically removed, in which, as the Bomberman makes progress by blowing up the walls in the path returned by that A*, the reward for that action would be greater.

In a complete learned program that is optimized for scoring the highest amount points possible, the way in which rewards are distributed when computing the Q-table in the learning process would also have to account for this scoring mechanism. Though this aspect of scoring is not implemented in the design, since the primary objective is to solve all variants by escaping the map, their framework for this implementation is partially developed, which can be further

built upon by placing preferences to actions that would result in these events though adjusting the weights and adding to the rewards.

Having considered the formalization of actions, defined the way in which discrete states of the world will be processed in identified, and defined reward weights, the Q-table can be constructed and one can begin populating the table with values. To start, any new or unrecorded state will be added to the table. The states will initially be assigned a Q value of zero for all actions in the action space, which will then be updated over time. To calculate the Q values, a Q-value iteration equation was used:

$$Q_{t+1}(s, a) = \Sigma_{s'} T(s, a, s') * [R(s, a) + \gamma * \max_{a'} Q(s', a')] \quad (1)$$

In this equation, the value is computed using the reward for an action and the discounted max Q value of the next Q value, as denoted by the multiplication with Gamma. However, this equation has the factor of randomness of taking an unintended action, represented by the transition factor T(s, a, s'). To simplify this equation since the desired outcome should be consistent, the probability factor was removed, leaving the equation:

$$Q(s, a) = r + \gamma * \max_{a'} Q(s', a') \quad (2)$$

As a result, the Q value calculation would no longer need to account for variable transitions between states. This equation is not yet complete, as there needs to be a factor of learning, which is captured by alpha, the learning rate. The following equation is the complete equation used to calculate Q values, while accounting for variable learning rates in the design:

$$Q(s, a) = (1 - \alpha) * Q(s, a) + \alpha * [r + \gamma * \max_{a'} Q(s', a')] \quad (3)$$

This equation is known as the exponential moving average, which is necessary to account for all of the updates to be made for the Q value depending on which action enters to which state (next) for a given state (current).

Reinforcement Learning Process

Active reinforcement learning allows for the character to pick the action as it explores the world. It is important to notice that the transition function T(s,a,s') and the rewards R(s,a,s') are unknown, the policy is stationary, and the Q-values are learned instead of the V-values. For active RL a set of samples represented by (s,a,s',r) allows to establish the rewards given at every state by the environment as mentioned in the previous section and shown by Figure 3 below.

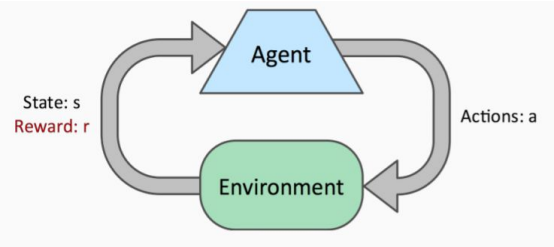

Figure 3: Reinforcement Learning General Idea Diagram[2]

Figure 4 below demonstrates an Agent that starts with an unpopulated Q-table (no states, no actions, and no Q-values) in Scenario 1 Variant 1-5. The hyperparameters where set to the following: learning rate of 0.8, discount factor of 0.8, and epsilon of 0. Rewards were unchanged as they are constants given by the environment. Figure 4 below shows averaged reward value for every 3 episodes, a total of 136 episodes. An episode in a representation of a complete game played, for bomberman a game starts with the character at position (0,0) and ends if the character reaches the exit, is killed by a bomb, killed by a monster, or max time is reached. After episode 49 the agent is able to learn the best Q-values, therefore giving it the best action based on the current state. Before episode 49 the rewards were negative. They seem to fluctuate around the -25 starting value. This behavior is interesting since the rewards at first start to increase and then decrease to below the -25 reward value. After that they start to reach the -25 value and once they do they immediately jump to the final positive reward of 50. A complete period allows for the agent Scenario 1 Variant 5 to learn from the environment the best Q-values for the states and determine the best action to take. After the best policy is determined, the agent is able to reach the exit 90 % of the time.

Note: 1 Episode = 3 Actual Episodes

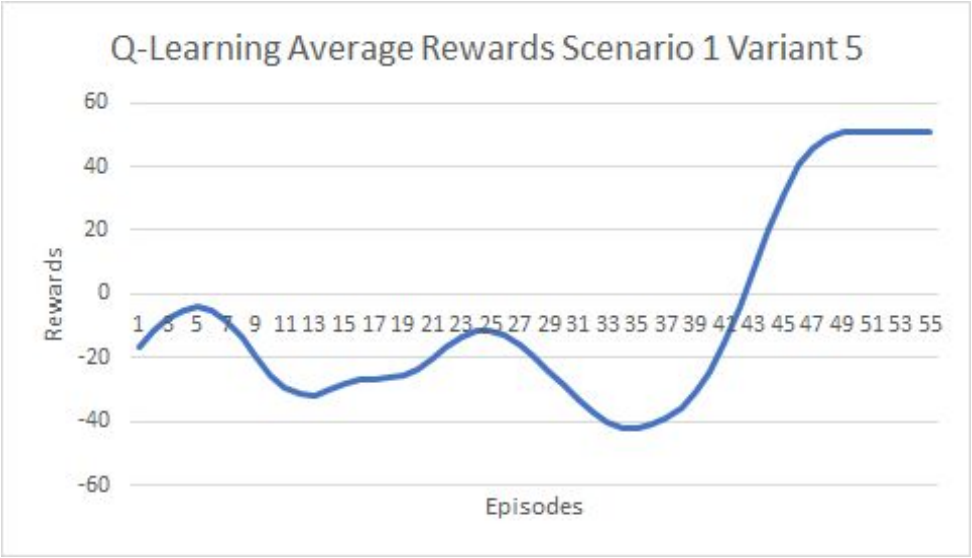

Figure 4: Q-Learning Average Rewards: Scenario 1 Variant 5
(alpha = 0.7, gamma = 0.8, epsilon = 0)

[2] Professor Calo Pinciroli, CS4341: Reinforcement Learning Lecture 8 Presentation Slide 10

Figure 5 below demonstrates an Agent that starts with an unpopulated Q-table (no states, no actions, and no Q-values) in Scenario 2 Variant 1-4. The hyperparameters were set to the following: learning rate of 0.8, discount factor of 0.8, and epsilon of 0. Rewards were unchanged as they are constants given by the environment. Figure 5 below shows a moving average of the reward value for a total of 1,300 episodes. After episode 573 the agent starts to learn the Q-values that give the higher rewards. After episode 959 the rewards begin to stabilize around 90. This demonstrates that the Agent is able to beat the highest variant on scenario 2 after training on every previous variant.

Note: 1 Episode = 2.25 Actual Episodes

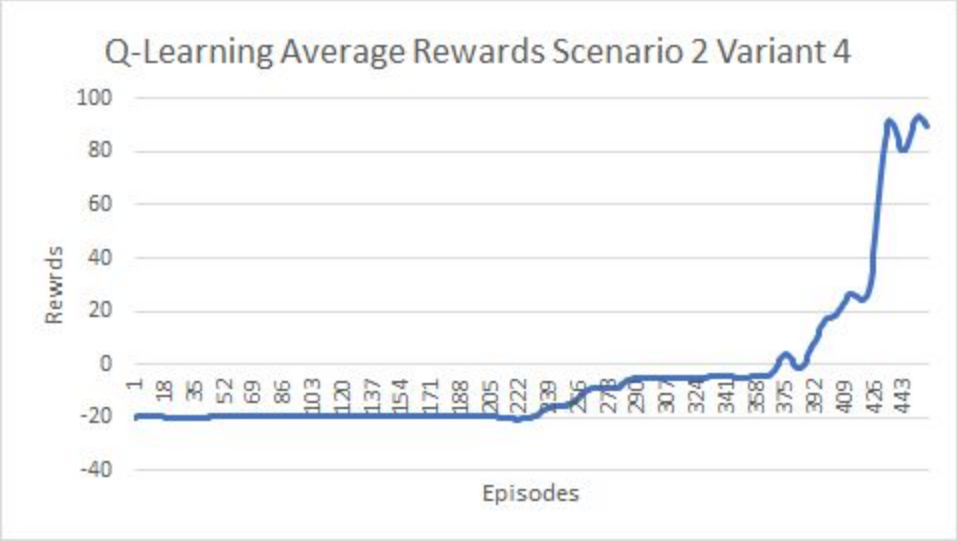

Figure 5: Q-Learning Average Rewards: Scenario 2 Variant 4
(alpha = 0.7, gamma = 0.8, epsilon = 0)

It is important to notice that the Agent was trained by going through each variant starting at one until it beats the variant multiple. This teaching worked for both scenario one and two. Scenario two had a longer training time than scenario one because of the amount of states created by different maze configuration.

Opportunities for Improvement

To address the project problem of solving multiple variants of different scenarios of the Bomberman game, a program that utilizes Q-learning along with A* pathfinding was developed, helping lead the Bomberman safely to the exit. While the implementation was successful in completing the objective, there are many opportunities to improve upon if not for the constraint of time.

Exploration and Exploitation

Picking actions at random can sometimes slow the processes of learning. Therefore a epsilon parameter is introduced between the values of [0,1] and a random value x between

[0,1]. The current implementation of Bomberman Q-learning described in this paper allows for the introduction of the epsilon value. This parameter is known to allow the algorithm to lean towards more of an exploration state or exploitation phase. The current Agent tends to exploit the best action known to it and based on that it is able to find the best actions based on it's states, but sometimes it is used to break loops during training. By adding the epsilon value this would allow for exploration and a possibly better set of action for every state.

A curiosity function would also be valuable as exploring randomly is inefficient. Curiosity would enable it to discover actions and states whose Q-values have not been established yet. A possible implementation with the current Q-learning Agent with an exploration function can be seen in equation 3 below highlighted in orange. The exploration function is represented by *f(U,N)*, where *U* is the utility estimate (Q-value) and *N* is the number of times an action has been tried.

$$Q(s, a) = (1 - \alpha) * Q(s, a) + \alpha * [r + \gamma * \max_{a'} f(Q(s', a'),N(s',a'))] \quad (3)$$

Regret

Regret allows to measure the total mistake cost and minimize it. The expected rewards have to be known to be able to calculate the difference between the given reward and expected reward. This would allow the trained agent to learn further as it makes mistakes even when it has already been trained. This could prove useful since the Agent theoretically would be able to reach the exit more times.

Approximate Q-Learning

In this implementation of Q-learning the world was generalized as described in the paper to discretized states. This feature representation could be further implemented by giving the features weights. The following equation demonstrates a possible linear value function that would allow to give features weights:

$$Q(s,a) = w1 * f1(s,a) + w1 * f1(s,a) + \ldots + wn * fn(s,a) \quad (4)$$

The initial weights assigned to equation 4 can be given at random as they are adjusted using equation 5 and 6 below. Delta calculates the error of the next Q-value with the rewards from the current Q-value. This means that if, in that state, something happens the feature is blamed by adjusting its weight. The only discrepancy with this is that since states are only represented by features in reality these states might have very different values.

$$\Delta = [r + \gamma * \max_{a'} Q(s',a')] - Q(s,a) \quad (5)$$

$$w_i = w_{i\ previous} + \alpha * \Delta * f_i(s,a) \quad (6)$$

Policy Search

The nature of solving the variants of Bomberman using Q-learning is difficult, since the number of states is significantly large due to the various combinations of character, monster, wall, etc. in any given cell. Taking into account how this implementation works, there is a lack of optimization in terms of deciding on the optimal move based on maximizing longer term rewards. This impacts the efficiency and effectiveness of the learning done as well as finding a solution. However, if there was some normalization made to the Q-table for tendencies of

certain actions between similar states or actions for such states that, on the Q-table, have similar value, the performance of the learning and finding a solution would be made more efficient. Incorporating a policy search to find an optimal policy will allow for the reduction of redundant or unnecessary movies being made, which would help with the learning scheme and potentially increase the speed at which a solution is found, thus providing the needed normalization of the Q-table.

Conclusion

To conclude, the integration using Q-learning and A* pathfinding successfully solves the escape mode of Bomberman by directing the Bomberman to exit all of the variants in the two scenarios. Q-learning was chosen for the flexibility of requiring no initially established model and the ease of implementation and use for learning to solve the different variants. The use of A* pathfinding helped expedite the learning process, as it was implemented to augment rewards for actions that follow the optimal path that was returned. The iterative evaluation of the active reinforcement learning done by the Q-learning implementation assisted in the selection of learning rate and reward discounting factors in the computation of Q values. Though the implementation solves the variants, there were many avenues of improvement to the developed program that can be explored. Opportunities include incorporating exploration versus exploitation factor in which the learning process can be manipulated to improve the rate at which a solution is determined (this could be done with Greedy Epsilon Alg. in which over time or number of episodes, the epsilon value increases and the program is more prone to select the best action based on the table), the implementation of regret to measure the total mistake cost and minimize it in order the Bomberman to more consistently reach the exit, building upon approximate Q-learning to initialize and update the weights in the linear equation, and to implement policy search to assist in normalizing the Q-table and improve optimal action selection over inconsequential states.