Bomberman Tournament

Mustafa Fuad Rifet Ibrahim

June, 2019

Contents

1 Introduction

This report is about the Bomberman Tournament that represented the final project to the Fundamentals of Machine Learning lecture of the winter semester 2018/19 of the University of Heidelberg. The task was to design an A.I. that could play a simplified version of the classic game Bomberman. The agents of the students were pitted against each other to determine the best agent. Among those agents three were picked for further analysis to demonstrate the variability of the approaches. The point of this report is to: (1) give a short summary of the three approaches that were picked, and (2) explore the range of possible augmentations to those approaches. However, this report is not meant to reflect the current state of possible augmentations and extensions to the described approaches but rather to give a small glimpse into the range of options.

2 Lord Voldemort

2.1 Summary

The agent Lord Voldemort was built with the Q-learning approach. Q-learning is a model-free apporach to learning an optimal policy that maximizes the total expected reward. The Q function $\mathcal{Q}_\pi(s, a)$ itself represents the exptected return if the agent starts in state s, executes action a and only then follows the policy π. The basis for obtaining an optimal policy is given by the following iterative update rule:

$$\mathcal{Q}(s_t, a_t) \leftarrow \mathcal{Q}(s_t, a_t) + \alpha \left[r_{r+1} + \gamma \cdot \max_a \mathcal{Q}(s_{s+1}, a) - \mathcal{Q}(s_t, a_t) \right]$$

$$\forall s : \pi(s) = \operatorname*{argmax}_a \mathcal{Q}(s, a)$$

r_{t+1} is the reward that the agent got when it transitioned from s_t to s_{t+1}, $\gamma \in (0, 1]$ is the discount factor which determines how much of the future reward estimation is taken into account and finally $\alpha \in (0, 1]$ is the learning rate which dictates how much of the difference between the old and new Q value is used for the update. The group decided to use the simplest implementation of that algorithm which is Q-tables:

	action1	action2	action3	...
state1				
state2				
state3				
⋮				

Table 1: Q-table

First, every entry in the table is initialized with zero and then through training all values are updated iteratively. In each update iteration the choice of the action can be made using a method like epsilon-greedy which balances exploration and exploitation. After choosing an action, that specific Q value is then updated according to the update rule stated above. The problem with this approach is the danger of the table ending up being too large. To solve this problem the group chose a smart construction of states[1]:

$$state = (Left, Up, Right, Down, Self, Self\_Bomb)$$

The $Self$ slot can hold three values: "Empty", "Bomb" and "Danger". The first two are used when the tile the agent is standing on is empty or has a bomb and the last option is used when one or more bomb radii intersect with the agent's tile. The $Self\_Bomb$ slot can hold two values: "True" and "False". These

are picked depending on whether or not the agent's bomb is on the field. The first four slots stand for the tile that is next to the agent in either movement direction. These can can hold a value representing the game object that is currently on that tile, e.g. "Bomb", "Crate", "Coin" etc., or hold the values "Empty" and "Danger" when no object is on that tile. Among those directions that hold the value "Empty" or "Danger" a special score, that depends on the relative distance to all objects on the map, is calculated and the tile with the highest score is assigned the value "Priority". The calculation of the score depends on custom weights the group chose for the different game objects. This state construction allowed the group to have outstanding performance with only around 3300 states in their final Q-table.

2.2 Possible Augmentations

2.2.1 Guided Q-Learning

This is an augmentation described by the group itself[1]. The idea is based on using human player data to speed up the training and maybe achieve a higher end performance. The training can be divided into four phases:

- Human player phase: The human player plays a certain number of games (preferably high number). During the human play, a Q-table is filled in the same manner as described previously for the standard Q-table approach, using the same state design, update rule and reward system. Essentially, the greedy epsilon method of choosing an action in a given state is replaced by human decision making.
- Agent phase: The agent plays a certain number of games, exactly following the approach described in the previous section. That means at the end of this pahse we have two Q-tables, one filled by human play and the other filled with the agent play.
- Mixing phase: Randomly selected entries from the human Q-table are inserted into the agent Q-table.
- Second agent phase: The agent trains the rest of the training period with the now augmented Q-table. The epsilon value may be resetted here to avoid overfitting to human data.

Possible variants of this augmentation are:

- Take out the first agent phase and the mixing phase and let the agent start it's training with the Q-table filled with human data as it's baseline.
- Be precise in the choice of Q-table entries that are transfered over to the agent Q-table. For example one could use the entries that have the highest Q-value among all the actions for the given state.

3 NOBEL

3.1 Summary

The agent NOBEL[2] utilizes a modern implementation of the Q-Learning approach by approximating the Q-function instead of using Q-tables. The idea is to use a neural network that, given a state, can return the Q-values for all actions:

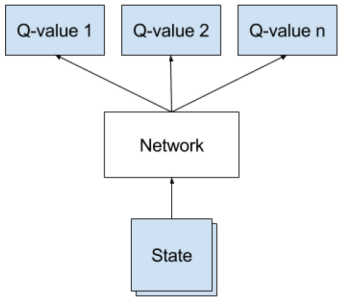

Figure 1: Deep Q-Learning Network[3]

The loss function used to train the network is the following:

$$\frac{1}{2}\left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right]^2$$

where r is the immediate reward received by performing action a in state s, γ is the discount factor mentioned in the previous agent description and s' and a' are the next state and the actions for that next state. The Q-values for the state s come from the first pass through the network and the Q-values for the resulting state s' come from the second pass. The main advantage, besides the amount of memory saved, is the fact that the neural network takes on the job of coming up with a good feature selection. Furthermore it also generalizes to unseen states. There are however a few problems with this approach that were addressed by the group, namely the instability of the approximation because of correlations between subsequent observations and correlations between the current Q values and the target values. The first problem was addressed by using the so called Prioritized Experience Replay. In this approach, minibatches of transitions that occured over the course of the training are picked based on a probability that depends on the temporal difference error calculated during update. This not only counteracts the problem of the correlation of subsequent observations but also speeds up training as more valuable experiences are preferred. The second problem was solved by using fixed Q-targets. In this approach two neural networks are used. The neural network that is used for target calculation has fixed weights for a certain number of steps while the other gets updated frequently.

In addition to this the group decided to implement another augmentation to the DQN approach that alters the neural network architecture itself, which results in the Dueling DQN architecture:

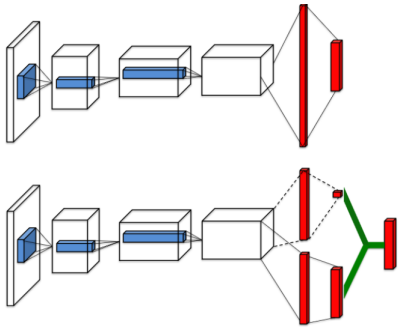

Figure 2: Dueling Deep Q-Learning Network[4]

After the convolution layers the network is split up into two streams. One stream outputs a single scalar value that approximates the value of the state that was fed in as input and the other stream outputs an advantage value for all actions. These outputs can be combined in different ways. The group chose to use the following formula as it provides good stability to the optimization:

$$Q(s,a) = V(s) + \left[A(s,a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a') \right]$$

This enables the network to approximate the value of the state and the advantage value of the actions independently of each other which leads to more efficient training and a higher end performance[4].

3.2 Possible Augmentations

3.2.1 Deep Q-learning from Demonstrations

Deep Q-learning from demonstrations[5] tackles the general problem of DQN algorithms to achieve a good policy only after millions of steps with poor performance. This is particularly problematic when applied to real world problems where agents have to interact with the real environment. Many such problems don't come with a simulator that represents a very good approximation of reality and so the agent has to start learning in the environment with an already high performance. In the specific case of the Bomberman A.I. this particular augmentation would have enabled the group to dramatically reduce the training time. In this DQN approach, the training is split into two parts, a pre-training phase where the agent learns from demonstration data and a normal training phase where the agent starts interacting with the environment, generating new

experiences and learning from a mix of those and demonstration data. Four losses are applied during the pre-training phase: 1-step double Q-learning loss, n-step double Q-learning loss, supervised large margin classification loss and L2 regularization loss. The Q-learning losses are used to ensure that the network satisfies the Bellman equation and is ready to be used in standard TD learning once the pre-training phase is over. The supervised loss is used to push the policy towards imitation of the demonstrator:

$$J_E(Q) = max_{a \in \mathcal{A}}[Q(s,a) + l(a_E, a)] - Q(s, a_E)$$

where a_e is the action the expert demonstrator took in state s and $l(a_E, a)$ is a margin function that is 0 when $a = a_E$ and positive otherwise. As can be seen from the equation, this loss leads to values of state-actions that have never been taken to be at least a margin lower than the value of the demonstrator's action. This is important because the demonstration data obviously covers only a small portion of the state space which would leave many state-actions to have no data to ground them to realistic values and using only Q learning losses would then lead to updates towards those unrealistic values. The reason for the L2-regularization is the prevention of over-fitting on the small demonstration dataset. All those losses are combined to form a weighted sum that represents the overall loss:

$$J(Q) = J_{DQ}(Q) + \lambda_1 J_n(Q) + \lambda_2 J_E(Q) + \lambda_3 J_{L2}(Q)$$

where J_{DQ} is the 1-step Q-learning loss, J_n is the n-step Q-learning loss and J_{L2} is the L2-regularization loss. Those losses are applied to all demonstration data in both phases but for self-generated data the supervised loss is not applied. One important thing to note is that once the agent starts to interact with the environment and generates transitions, these transitions are stored in the same replay buffer as the demonstration data and once that buffer is full the oldest self-generated transition is over-written. Demonstration data stays in the buffer permanently. Furhtermore when using prioritized experience replay, small constants ϵ_a and ϵ_d are added to the priorities of self generated and demonstration data to allow for control over the relative sampling of those two types of data, specifically the probability of demonstration data is increased.

3.2.2 Opponent Modeling

Opponent modeling[6] is a useful augmentation to DQN in scenarios in which the agent has to play against or together with other agents that do not behave strictly in a deterministic way. This means the other agents can not be simply viewed as part of the environment and their policies have to be considered explicitly. In this way potentially bad strategies of opponents can be exploited if that bad behavior is predicted. This augmentatation is thus particularly useful

for Bomberman and therefore represents another possible option to extend the DQN approach. The optimal Q function

$$Q^*(s,a) = \sum_{s'} \tau(s,a,s') \left[r + \gamma \max_{a'} Q^*(s',a') \right]$$

, where $\tau(s,a,s')$ is the transition probability i.e. obtaining state s' when executing action a in state s, now has to be defined relative to the joint policy of the other agents:

$$Q^{*|\pi^o} = \max_{\pi} Q^{\pi|\pi^o}(s,a)$$

, where o stands for the joint action of all other agents. In their paper, He He et al propose a network structure called Deep Reinforcement Opponent Network (DRON) that models $Q^{\cdot|\pi^o}$ and π^o jointly. It consists of a standard Q network that does action-state evaluation and an opponent network that learns a representation of π^o. There are two options for combining those two networks: DRON-concat and DRON-MOE. The first option simply concatenates the output of two networks that take features of the state and features of the opponent. This concatenation then predicts the Q value. The second option uses a Mixture-of-Experts network to explicitly model the opponent action. Every expert network predicts a reward for the current state and combining the predictions from all those expert networks gives the final expected Q value.

4 LaranTu

4.1 Summary

The third agent in this report was built with an entirely different approach. Instead of using Q-Learning this group decided to use a Monte-Carlo Tree Search based approach[7] very similar to AlphaZero. Their approach can be divided into two main parts:

- Rule based/heuristics
- Monte Carlo Tree Search guided by a neural network

They used the first part, the heuristics based approach, to govern the behavior of the agent whenever it wasn't close to any other agent. For this, they represented the distances to nearby agents using weighted graphs and utilized available fast algorithms to calculate the shortest path to an agent. Obviously this part of the algorithm dominates the early game as all agents are far away from each other. The second part takes over whenever the agent gets into a certain range to other agents. In this case a mini-game is created that only includes the agents close by and no coins. That means the game is transformed to a simpler perfect information game version with the only objective being to kill the opponents. This mini-game represents the game on which the Monte-Carlo Tree Search is applied. The group utilized self play during training and so the tree structure for a given mini-game would have the form:

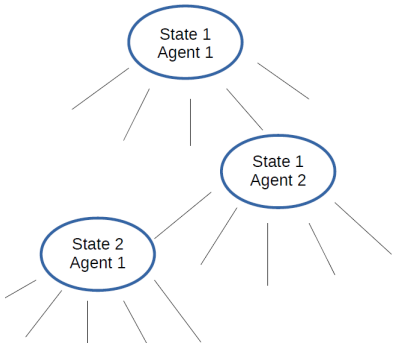

Figure 3: Tree Structure For LaranTu

This is the second simplification of the game. Through using this particular tree structure the group viewed the game as a turn based game. The tree search itself was guided with a neural network instead of the classic approach of playouts and the standard selection formula. The network predicts a probability of success for the actions (policy part) and a value for the current state (value part). Both outputs are used to guide the tree search in the selection phase of MCTS. The basic iteration during training would look like this:

- Create mini-game for all groups of agents
- Execute Monte Carlo Tree Search for a certain number of repetitions
- For the part of the tree that corresponds to the highest visit count move, repeat Monte Carlo again. This effectively means that the mini-game is played out before any move in the main game is made. This however goes on only for a limited amount of mini-game steps.
- At the end of the mini-game use the move with the highest visit count as the ground truth for the policy part of the network and the end result of the mini-game as the ground truth for the value part of the network.
- Use the action with the highest visit count as the action that should be performed in the main game for the given agent

The choice to use mini-games not only made the task simpler but also enabled the group to update their network in every step of the main game. This means that an increasingly better version of the network was used to guide the following MCTS. Another advantage is the fact that the tree search returns an improved recommendation of the next move at each iteration. This however also led to this agent being very slow compared to other agents and so it always received a penalty.

4.2 Possible Augmentations

4.2.1 MCTS Modifications

One possibility for improving the group's approach is to specifically tackle the MCTS algorithm itself and try to apply various extensions and modifications that have been developed over the years. One option is to use domain specific knowledge. This can be incorporated into the MCTS through for example these two different ways:

- node selection
- pruning

Node selection can be altered through assignment of nonzero priors to the value estimation or the score when creating each child node, effectively influencing the frequency with which the children will be selected[8]. Another method that influences the node selection is more dynamic in nature as the influence of the domain specific nature decreases over the number of times the given node was visited[9]. This is done through adding an extra term:

$$\frac{b_i}{n_i}$$

, where b_i is a heuristic score of the i-th child node and n_i is the number of times the child node i has been visited so far. The idea behind this is to give

starting help with domain specific knowledge but then as the move selection through the tree search gets an increasingly better approximation of the best move, enough iterations of the tree search will have been executed and so the selection based on the score without the domain term can be trusted more. The second way of bringing in domain specific knowledge mentioned above relates to the process of ignoring or deleting certain parts of the whole tree as they are deemed irrelevant. In this case the process of pruning a part of the tree is based on domain specific knowledge. For example in the specific case of Bomberman certain situations could be branded as hopeless and therefore no time should be wasted exploring all the different ways of playing out these hopeless situations. Other modifications to the MCTS algorithm that may be considered involve the way the "best" child node is defined when actaully selecting a move to be played in the main Bomberman game after finishing the last MCTS iteration. The group chose the child node with the highest visit count but there are other possibilities:

- the child with the highest value
- the child with both the highest visit count as well as highest value. If there is no such child node at the moment then more iterations of the MCTS are executed until such a situation occurs.
- the child that satisfies some kind of lower confidence bound

It might be worthwile to test for signifcant differences in performance when choosing these different approaches to final move selection.

5 Conclusion

As mentioned at the beginning of the report, this report clearly does not exhaust all the possibilities to augment the approaches used by the student groups. Nonetheless, through this little exploration it hopefully became clear that the amount of options is vast and that Bomberman is a fitting game for trying out many different approaches.

6 References

[1] Dan Halperin and Ella Stahl: Report - Lord Voldemort
[2] Tatjana Chernenko, Jan-Gabriel Mylius and Valenti Wüst: Report - NOBEL
[3] https://neuro.cs.ut.ee/wp-content/uploads/2015/12/dqn.png
[4] Wang et al: Dueling Network Architectures for Deep Reinforcement Learning. 2016
[5] Heste et al: Deep Q-learning from Demonstrations. 2017
[6] He et al: Opponent Modeling in Deep Reinforcement Learning. 2016
[7] Anna Sommani and Peter Sorrenson: Report - LaranTu
[8] Gelly and Silver: Combining Online and Offline Knowledge in UCT. 2007
[9] Chaslot et al: Progressive Strategies for Monte-Carlo Tree Search, 2008