Using Reinforcement Learning to Play Bomberman

Vasil Manev

April 2022

1 Introduction

This report is part of the final project assigned by Prof. Ulrich Koethe for the lecture "Fundamentals of Machine Learning" at the University of Heidelberg in the winter semester 2021/22. Here I will explain the main techniques I considered, used, and finally submitted to try and solve the problem at hand - creating an agent capable of learning how to play the classic arcade game "Bomberman". The agent's name is "Deeprest Bot" and its performance explains it all.

2 Researching different approaches

In this section I will give an overview of the approaches I researched and experimented with.

2.1 Monte Carlo Tree Search

The Monte Carlo Tree Search (MCTS) algorithm is a very popular algorithm used in various software as a solution to the game tree of a specific game. A game tree represents all possible game states in the form of a graph. Usually we would use game trees with perfect information at hand, meaning that board games, for instance, are easily represented by game trees. It comes as no surprise that MCTS is employed in many approaches to tackling games such as chess, checkers and Go. So why use it here? At its core, the game "Bomberman" is a sophisticated board game. Thus, I assumed that the multitude of game states could be represented as a game tree.

How MCTS works?

The essence of the MCTS algorithm is to find the leaf of a tree by taking a path that goes along the best child nodes. To achieve this, we have to employ some paradigm and the best practices I have found are all using the upper confidence bound for trees or UCT:

$$\frac{w_i}{n_i} + C * \sqrt{\frac{t}{n_i}}$$

Where the first fraction represents the win ratio by dividing the number of wins w_i after the i-th move over the number of simulations after the i-th move. C is what is known as the exploration parameter and is equal to $\sqrt{2}$ as employed in the UCB1 algorithm. Under the square root we have t which is the number of simulations for the parent node. When UCT can no longer be used to find the successive node, MCTS will simply append all possible states from the leaf, basically expanding the tree. The next phase includes choosing a new child node from which to simulate the game until reaching the resulting game state. This happens arbitrarily. Once the game is over, that is, someone wins, the algorithm will go back to the root and will update the visit score of all nodes along this path. Win score is also updated if for each node v_i if the player at that position won the game.

Upon understanding the algorithm I now had to figure out a way to implement it. First I tried it out on a simple game - tic-tac-toe. The results were pretty charming, given that tic-tac-toe has a total of 3^9 or roughly 20,000 possible states and only 5,478 of them actually work in-game. This is where I realised that the full playing field in "Bomberman" might be a bit too large and I would have to break down the game into parts where I would implement the algorithm. The main idea would have been to recognize some level (i.e sub-game) at which I would achieve a perfect information state, so maybe focusing on a 3x3 or a

4x4 neighbourhood and aiming to simply kill enemies or collect coins. This is where the MCTS would have potentially taken place for the most part, forcing the agent to treat the game as a collection of sub-games to be won. This is where I encountered the very first set of problems that were bound to come my way. It would suffice if I were to say that I had no idea how I would code this efficiently, however, that was a concern for later. The backpropagation of feedback towards the root seemed inconceivable given that I would have to take care of multiple sub-games at once. Some optimization was needed but I was not entirely sure how I would do that and so I decided to get back to reading.

2.2 Q-learning and Deep-Q-Networks

After a number of unsuccessful attempts at integrating the simple MCTS code I worked out into the architecture of this project I decided I would start looking for classical solutions to video game problems such as the one at hand. One of the most popular choices for similar problems seemed to be Q-learning. But what is that? Q-learning is an RL algorithm devised back in 1989. It is an approach to solving the Bellman equation for Markov Decision Processes:

$$q_*(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma \max_{a'} q_*(s',a')]$$

The Q-learning approach employs the Q(s, a) action-value function and it looks like this:

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[R_{t+1} + \gamma \max_{a} Q(S_{t+1},a) - Q(S_t,A_t)]$$

So, what do these equations tell us? Well q_* is optimal action-value where as q and Q are the real and estimate action-value respectively. The quantity in the square brackets of the action-value function represents the error term and shows us how far off target we are. The term R_{t+1} represents the reward at timestep $t+1$. The environment transition dynamics are represented by $p(s',r|s,a)$. The discount rate is the way our algorithm treats immediate and current rewards. The discount rate γ is defined in this interval: $0 \leq \gamma \leq 1$. What Q-learning does is it makes two approximations. The first one is to replace the expectation value from the Bellman equation with a sample of estimates. Sampling is needed since Q-learning is model free i.e it does not access the model of the environment dynamics. The second approximation is based on temporal difference - Q-learning will replace the target from the Bellman equation with the one-step difference from the Q(s, a) action-value function. This method's reduction of variance comes at a price - using an approximate function in the target introduces bias.
Deep-Q-Networks
I still wasn't happy with what Q-learning had to offer - the static Q-tables seemed like they could use some improvement. My research led me to a paper published by DeepMind titled "Human-Level Control Through Deep Reinforcement Learning". The paper gives a detailed yet intuitive explanation of how

Deep-Q-Networks can be used to improve on Q-learning. It gave me insight and clarity into the problems that I myself had already partially recognized. What solutions did I find that made me go for DQN in the end? The introduction of the experience replay buffer and sampling of minibatches in order to improve the stability of the algorithm was quite convincing. Here I have to admit that I saw no problems with training on correlated sequential date because for one - I didn't notice that initially. Also, the bias introduced by temporal difference methods goes hand in hand with another problem - moving targets. The authors of DQN came up with a great solution - freezing the target network. This ought to work.

3 Introducing Deeprest Bot

3.1 Main Idea

My research was still on-going. I have to admit that at the beginning of the project I didn't feel up to the task to solve this problem but was excited to learn. To my surprise a lot of DQN implementations that I could find online employed convolutional neural networks as the base for their DQN. This means that the agent would learn to play from the pixels on the screen rather than something else. So I decided I would devise a convolutional network to try and train my agent to play "Bomberman".

3.2 Build

I employed a very classic build for my network:

```
class DQN(nn.Module):
    gamma = 0.7
    learning_rate = 0.004
    def __init__(self, dim_out):
        super(DQN, self).__init__()
        self.model_sequence = nn.Sequential(
            nn.Conv2d(3, 32, (5,5)),
            nn.LeakyReLU(),
            nn.Conv2d(32, 32, (5,5)),
            nn.LeakyReLU(),
            nn.Conv2d(32, 32, (3,3)),
            nn.LeakyReLU(),
            nn.Conv2d(32, 1, (3,3)),
            nn.LeakyReLU(),
            nn.Flatten(start_dim = 1),
            nn.Linear(25, 128),
            nn.LeakyReLU(),
            nn.Linear(128, dim_out),
            nn.Softmax(dim=1)
        )
        self.loss = nn.MSELoss()
        self.optimizer = optim.Adam(self.parameters(), self.learning_rate)
        self.device = T.device('cuda' if T.cuda.is_available() else 'cpu')
        self.to(self.device)
```

Where my implementation differs from most DQN implementations I encountered during my research is that I set a fixed learning rate and employ no ϵ-decay function. However, I trust the Adam optimizer to compensate for that. I also included an option for the network to be trained on a GPU. After making sure that I have set up my network properly I had to introduce two further functions. The forward function that predicts the next action and the backward function (here 'train bot') that trains the function approximator. The 'train bot' function employs the Q-learning strategy through the most important values - the action value of the current state, the action value of the next state and the action value of the expected state which goes by the formula $q_{s+1}\gamma + R$, where q_{s+1} is the next state action value, gamma is the discount factor and R is the reward. The loss function in the 'train bot' function employs the nn.MSELoss() function and its parameters are the action value of the current state as well as the action value of the expected state.

3.3 Training and Problems

After successfully setting up the network i.e making sure all multiplications were feasible and my code compiled without error it was now time to try it out against the rule based agents. This is where trouble began. I trained my agent for the default number of cycles without using the –no-gui functionality and was immediately alarmed as the program started returning [INVALID ACTION] statements. I deleted the saved model and re-initiated training, this time with GUI on. My agent would wander one or two spaces up and down, stand still and then blow itself up. Sometimes it would manage to plant a bomb and avoid it but that was more or less an exception to the rule. It barely moves and then kills itself - now what? I was first interested why the log would show me countless invalid actions and no events of the type 'waited' - I could see the bot not moving. I started tweaking whatever I could - rewards, convolutions - but to no avail. The most 'progress' I managed to achieve was to replace all invalid actions with waits. That's when it dawned on me. The bot is trying to move through walls and that is causing the 'invalid' statements. Sadly, this insight did not help me to devise a solution to this problem. Notably, training the bot less gives it a bigger chance of doing something in game than training it more. After a 1000 training cycles it just stands still. However, after 10 to 20 cycles it would manage to move a couple of times in-game and then blow itself up. This is why I named it 'Deeprest Bot'. I would have loved for the name to mean that my agent takes its time before making an elaborate decision for its next couple of moves, but I must admit that the name is just a play on the word 'depressed'. Sadly, my implementation converges to a suboptimal solution to the problem at hand.

4 Conclusions

I cannot be sure as to why my version of DQN did not deliver in this scenario. Could it be that convolutional networks are not well fitted for this game? Did my bot simply fail to make a distinction between the available playing field and the brick walls? Is my skill not enough? Although I deem my code and agent a failure, I am hopeful that by submitting my solution I will get some insight on why and where I failed and perhaps would even receive useful feedback on how to fix my network and have it running and training my agent properly. I found this project to be a teachable moment and was humbled by the challenge that I was presented with.

Sources

https://storage.googleapis.com/deepmind-data/assets/papers/DeepMindNature14236Paper.pdf
shorturl.at/bxBS2
https://towardsdatascience.com/monte-carlo-tree-search-implementing-reinforcement-learning-in-real-time-game-player-a9c412ebeff5
https://paperswithcode.com/method/dqn https://arxiv.org/abs/1312.5602v1
https://www.jstor.org/stable/24900506

Github Repository

https://github.com/vasilmanev/BMBRMN-final