Praktikum aus Künstlicher Intelligenz
Lab Report of Group C

Jinyao Chen,[1] **Zuojun Shi,**[1] **Qiao Sun,**[1] **Zhiyang Tong,**[1] **Siyang Zhang**[1]

[1]TU Darmstadt

jinyao.chen@stud.tu-darmstadt.de, zhiyang.tong@stud.tu-darmstadt.de, zuojun.shi@stud.tu-darmstadt.de,
qiao.sun@stud.tu-darmstadt.de, siyang.zhang@stud.tu-darmstadt.de

Abstract

Describe your chosen approach and give the key finding(s) of your work.

Introduction

The Pommerman benchmark aims to provide a test environment for the research of reinforcement learning of multi-agent. The environment is built based on a popular Nintendo game, Bomberman, in which killing others using bombs and being the last one staying alive is the main goal. Bombers need to execute one action at each step, including moving actions like UP, DOWN, LEFT, RIGHT and STOP, as well as placing a bomb at the position of itself, which is only allowed when its ammo is greater than 0. A board will be randomly created at the beginning of each round, consisting of wood, which could be destroyed by bombs, rigid, which is unaffected from bombs, passage, where bombers can move into, and Flame, which is the product of the blast. Additionally, there are powerup items either appear in the passage, or hide behind the wood. Three types of powerup are ExtraBomb, increasing bomber's ammo by 1, IncrRange, increasing bomber's blast strength by 1, and Kick, allowing the bomber to kick bombs. All bombers are initially placed at a fixed position and have ammo of 1 and blast strength of 2. A bomb blasts 10 steps after it is laid and produces flame for 2 steps, with the range of the bomber's blast strength at four directions, destroying bombers as well as powerups and woods. A bomber agent being killed will not be allowed to act any more and disappear on the board.

More challenging is the task for our lab, that we play in the radio team mode. The competition involves 4 bomber agents and each two of them playing as a team, running on a 11x11 board with partial observability of 4 blocks in all directions. When the game starts, the diagonal agents become teammates and the team with both of the bombers being killed is lost. Unlike normal team mode, radio mode allows agents to send a message of two integer value to their teammates at each step. To avoid tie games, the time horizon is increased into 800 steps.

As the solution to the advanced game rule, we apply Deep Q-Networks algorithm(DQN) to the training and design our own reward setting. To achieve more efficient and stable training results, we also implement some extensions of DQN into our code, as well as the strategy of action filter.

Related Work

Deep Q-learning Since that Q-Learning (Watkins and Dayan 1992) is poor at training of large state space, Deep Q-learning (Mnih et al. 2015) has been proposed to solve this problem. A convolutional neural network is used to estimate the values for each action for the given state. With a certain learning rate t+1, using the network on state St, action At is selected greedily and the corresponding rewards Rt+1 and the state for the next step St+1 will be output by the environment. Instead of the Q-table in Q-learning, an experience pool called replay memory buffer (Lange, Gabel, and Riedmiller 2012) can store millions of transitions (St, At, Rt+1, t+1, St+1), from which memories are randomly sampled from the buffer to break the correlation between them. The network architecture contains two part: A value network and a target network used to choose action, whose parameters are updated at certain steps. The value network is optimized by minimizing with stochastic gradient descent the loss:

(formula),

where t is the step count for the sampled data, theta and theta- refer to the value- and target network.

Rainbow DQN. (Hessel et al. 2018) After examined several extensions of DQN algorithm and Rainbow DQN integrated them all together in their agent. The algorithms are proved to get better performance and data efficiency in combination. The study also showed their individual contribution to the final performance. However, since the algorithms are not applicable for all types of games, we only choose those that fit our task.

Double-DQN. As an off-policy learner, the original DQN can easily get overestimated results. To overcome this problem, the on-policy learning Double DQN (Hasselt 2010) updates its Q-value using the next state s and the current policy's action a instead of the action a' chosen from the target network. The loss

(formula),

optimizes the value network, avoiding overestimated

Q-value and thus improving the training.

Prioritized Replay Buffer. The randomly sampling in original DQN doesn't consider their correlation with priority. The prioritized experience replay buffer (Schaul et al. 2015) orders the samples by TD error, giving samples where there is much to learn, a higher probability to be selected, thus enabling a more efficient training.

Dueling-DQN. Unlike the Q-value directly output by convolutional network, Q-values in Dueling-DQN (Wang et al. 2016) are defined by both value- and advantage function, which enables to converge to be faster. The Dueling-DQN architecture is combined with two streams, where one receives only state information and estimates general value for states, the other gives the advantages for each action. The formula of Q-value is then:

(formula),

where , , and are parameters of the networks.

Multi-step learning. In the early phase of training, the deviation of the obtained target value is large, which makes it take more time to estimate precise value. In Multi-step learning (Sutton 1988), rewards could accumulate for a few steps as

(formula),

The loss is then

(formula).

When the step number n is reasonably selected, the learning speed could be greatly improved.

Noisy Nets. To improve the agents' exploring ability, DQN uses the strategy of greedy to select actions. Another method is NoisyNet (Fortunato et al. 2017) that adds noise to the network to achieve the same effect. When the standard linear y = b + Wx is changed into:

(formula),

the network could ignore the noisy in the later episodes with self-annealing.

Approach

In this section, we will first describe the challenges we meet in the multi-agent learning of pommerman benchmark, then propose our own approach and explain why it could solve these problems.

Challenges

Numerous state information and non-static game map: In pommerman environment, an agent receives its individual observation at each step, which consists of a 2D array of map elements with their coordinate, and agent's information including position, teammates, bomb strength, ammo, and whether it can kick bomb. Note that a map containing all elements would be difficult for the agent to learn, because they would be implicitly assume to have linear connection between each others, which is actually not the fact. The game map is non-static, generated randomly at the beginning of each round, which means that there is a huge amount of probabilities of the map for the agents to learn.

Vague rewards from the system: Although the system also gives rewards to the agents, they are almost unusable. A binary number would be output only at the end of the game for each agent, 0 for lost and 1 for win, while a tie game would not give any rewards. The agents could hardly learn something at a single step, and since that a round sometimes need hundreds of steps to end, the final rewards couldn't affect so much on the early steps.

Partial observation: In radio team competition, agents are not able to see the whole map, but only a range of 4 blocks around itself. The lack of the information about bombs, agents, or powerup makes the learning difficult, because agents are not able to escape from bombs outside the observation. Even the SimpleAgent, the heuristic agent playing against us during training, are often killed in this mode and the winning becomes meaningless.

Our approach: Deep Q-learing can predict value of big action space and has achieved great progress in Atari games, an advanced approach will be applied to our task. Unlike the Atari games, the screen image of Pommerman consists of separate elements, so the board from observation will be preprocessed into multi-channel to break their correlation, and board information will be treated differently as the agent's state information. The training is carried out in our manual environment, where only the trained agent has partial observation, to ensure the opponent agents not being killed so easily. The reward shaping could promote the progress of reinforcement learning, but the simple reward given just for the wining is not enough. We will implement more detailed rewards, that could lead to the winning, for the agents. Another difference between Pommerman and Atari games is, that the actions are not always executable. In traditional DQN, the non-executable action will increase the amount of meaningless learning. To handle this, an action filter will be added to the action selection, preventing agents selecting non-executable actions and speeding up the learning progress. Additionally, some extensions of DQN algorithms were implemented, and we examined their effect on the efficiency and agent's performance. Those getting good scores would be applied to our architecture.

Technical Details

Our model is based on DQN Algorithm and improved by different components. Figure 1 shows the architecture of the framework. In this section, the components will be introduced in detail and how they are working together.

Data pre-processing

The observation data for each agent is automatically returned by the environment after it receives agents' actions as input. To transform the data into a form that could be computed by the network, a pre-process sis necessary. Table

Info	Value or Numer in board
Path	0
Rigid	1
Wood	2
Bomb	3
Flame	5
Power up	6, 7, 8
Bomb blast strength	2D Integer Array
Bomb life	2D Integer Array
Bomb moving direction	2D Integer Array
Flame life	2D Integer Array
Ammo	Integer ≥ 0
Blast strength	Integer ≥ 2
Can kick	0 or 1

Table 1: Behaviours and the corresponding reward

1 is a list of information given by the observation. Since that the networks could give linear correlation for the different types of the elements, to avoid this, we divide the map into 11 2D arrays, which has 1 at the position where the element is, and 0 for nothing. Figure 1 is a simple example of how we divide the map. The other one-dimensional information is convert into 2D, so that all data has the same shape.

DQN Network

To apply DQN to our task, we build a network inside the agent to predict action for the giving observation. During the training, after the environment is started, the parameters of the network will be initialized and a capacity of 100,000 memory replay buffer will be created inside the agent. At each step, the transitions (St, At, Rt+1, St+1) is added into agent's buffer, which corresponds to the preprocessed observation of the current step St, of the next step St+1, the selected action At, and the computed reward Rt+1. Before sampling, the transitions in buffer are first sorted by their TD-error. A size of 256 batches are then sampled from the buffer to optimizing the network. At the early phase of training, we pack several steps together and compute the total reward as a sample, and switch to double steps later. The target-net's parameters are replaced every 20 steps with the eval-net's. The Q-values are predicted by target-net with actions that are selected by eval-net, using the idea of Double-DQN. The pseudocode 1 shows the process of the training in one episode.

Figure 2 is the architecture of the network, built with three convolutional layers and two linear layers. We also implemented the dueling network to compute state value with a value network and the value brought by the actions with an advantage network. Additionally, we use Noisy-net to enhance agent's exploration ability, by adding Independent Gaussian noise to the dense layers.

Reward Shaping

A simple reward for win or lost is not enough for our DQNAgent to quickly learn usable strategy, for that we set a detailed reward for different behaviours, which are listed in Table 2.

Algorithm 1 Training code in one episode

```
while $notfinish$ do
    $S_t \leftarrow$ preProcess($obs_t$)
    $action \leftarrow$ selectAction($S_t$)
    $next_observation, result, finish \leftarrow$ Env($action$)
    $S_{t+1} \leftarrow$ preProcess($obs_{t+1}$)
    $reward \leftarrow$ rewardShaping($S_t, action, S_{t+1}, result$)
    $agent.buffer$.add()
    $agent$.train()
```

In terms of game results, we set the reward for winning a game to 0 to avoid the negative effect brought by the winning because of enemies' suiciding. For losing a game, we set value -1 in order to give enough punishment.

Strategies like blasting a wood, picking up a powerup and kicking a bomb, we set values respectively to 0.01, 0.05 and 0.02. A reward for laying a bomb is -0.005, as it is unsafe when an agent lays a bomb that could potentially can kill itself. Reward for laying a bomb and making an enemy stay in the blast range is 0.2, but doing the same thing to the teammate, reward will be -0.1. When the distance between the agent and a bomb is enlarged in the next step, it will get 0.005 reward and the reward a decreasing distance is -0.01.

Considering of the moving action, we set the reward for taking a stop action to avoid the flame or bomb to 0. We set a reward for moving to 0.001, which means that the agent really moves and doesn't knock into the wall. If the agent doesn't encounter a bomb but stands for nothing, we set the reward for staying to -0.003. A reward for taking an action, but hitting at wood is set to -0.01. If the agent's reactive actions are meaningless, for example, repeating action 1, 2, or action 3, 4, in this condition, the reward is -0.005. When the agent is stuck at a dead end where it is surrounded by wood and will be killed by its own bomb after several steps, the reward is -0.1. A reward for ignoring a powerup or not bombing a wood is set to -0.0015. Here we mean that the agent sees wood but doesn't try to bomb it, or it sees a powerup, but doesn't pick it.

Action Filter

The main idea of the action filter is to filter the behaviour according to the security level. The safety level can be divided into four grades: 0 is safe, 1 is dangerous, 2 is high risk, and 3 is risk of death.

The security level is set through the following two steps. First we check the 6 action feasibility: If it is a move behaviour, and feasible, add it to the move list. If it is the act of placing bombs, and the bombs are currently available and meaningful (can be bombed to wooden walls or enemies), add this act to the mobile list. Then we set the safety factor, as shown in the Table 3: If a bomb was placed in the previous act, the act of entering a dead end is classified as dangerous level 3. If there are no bombs around, check whether the movement behaviour in the list will enter the fire, if so,

Behaviour	Reward
Win	0
Lose	-1
Bomb wood	0.01
Pick powerup	0.05
Kick	0.02
Lay bomb	-0.005
Lay bomb near enemy	0.2
Attack teammate	-0.1
Get away from bomb	0.005
Get close to bomb	-0.01
Stop in front of bomb	0.001
Move to another position	0.001
Stay at its position	-0.003
Move towards wood	-0.01
Move to dead end	-0.01
Move back and forth	-0.005
Ignore wood or powerup	-0.0015

Table 2: Behaviours and the corresponding reward

it is classified as a danger level 3. If the agent is within the bomb range, the following judgments are made: There is a flame with life greater than 0 or the remaining time of the bomb is equal to 1 on the coordinates after the movement behaviour, which is classified as a level 3 danger. The remaining time of the bomb are greater than 1 and less than 4 on the coordinates after the movement behaviour, which is classified as a danger of level 2 or higher. The remaining time of the bomb is greater than or equal to 4 on the coordinates after the movement behaviour, which is classified as level 1 danger or higher.

The final returned action will be selected through the following steps:

1. Find the behaviour security level of 0 in the move list and put it into the action list. If the original behaviour is in the action list, return the original behaviour;
2. If there is no safe behaviour, find out the level 1 behaviour and put it on the action list. If the original behaviour is in the action list, return the original behaviour;
3. If the action list is empty, find the level 2 action and put it on the list;
4. If the action list is still empty, find the level 3 behaviour and put it in the list;
5. If the action list is still empty, return a random behaviour;
6. If the action list is not empty, the behaviour other than the stop behaviour will be randomly returned first, if not, the stop act will be returned.

Data Argumentation

By slightly modifying existing data to increase the amount of data, Data Argumentation could prevent the model being

Level	Behaviour
3	Lay a bomb and move into dead end in the next step
3	No bomb nearby and move into flame in the next step
3	In blast range and move into flame in the next step
3	In blast range and bomb blasts in the next step
2	In blast range and bomb blasts in the next 2-4 step
1	In blast range and bomb will not blast after 4 steps

Table 3: Levels of the agents' dangerous behaviours

overfitting. Considering the shortage in our training algorithm that agent's start position is fixed, so that agents could have the same performance starting at different position, we learn from the idea of Data Argumentation, generate fake samples as if agent has been trained at the other corners.

Figure 2 is an example of how we generate them with the original observation.

Custom Environment

The SimpleAgent is the type of agent that we used as opponents in training, having some built-in methods that enable it to do some clever behaviours like escaping from the attack of bombs. However, we observed that the radio competition mode(v2 mode) has the imperfection that the SimpleAgents are easily set them in a situation where no method could help them escape, which leads to a suicide. In order to avoid those cases that are of no worth to learn, after enough amount of experiments, we designed a custom game environment for training, Where agents have partial observation as v2, and are able to send messages to their teammates.

Evaluation

Give e.g. ablation studies, results against agent heuristics, results from the tournaments, progress of your agent's performance during the lab ...

Expriments

During the lab, we have been continually improving our agent to achieve better performance and win rate. Since the python environment that we use is lack of fast running speed, the early experiments were carried out in one vs one mode, as it has simple map and considerable running speed. When the basic architecture and related techniques were determined, we then moved to FFA and radio mode to work on more adjustments.

Our experiments are divided into training part and final performance part. In the training part, we record the agent's accumulated reward within each game round to see how it changes with the increase of training episodes. The final performance part is the test of the trained models, counting their win rate in 10000 rounds. The enemy agent in both one vs one mode and radio competition mode is the SimpleAgent, while the teammate agent during training is the RandomAgent, and is set to our DQNAgent in the performance test.

Results

In early experiments, the agents could hardly do some smart behaviours after thousands of episodes, with a training architecture that only includes a basic DQN Algorithm.

Conclusion

Briefly summarize your approach and give information about its performance. What is/are the message(s) of your work for the reader? How could one extend your approach? What could be interesting future work?

References

Fortunato, M.; Azar, M. G.; Piot, B.; Menick, J.; Osband, I.; Graves, A.; Mnih, V.; Munos, R.; Hassabis, D.; Pietquin, O.; et al. 2017. Noisy networks for exploration. *arXiv preprint arXiv:1706.10295* .

Hasselt, H. 2010. Double Q-learning. *Advances in neural information processing systems* 23.

Hessel, M.; Modayil, J.; Van Hasselt, H.; Schaul, T.; Ostrovski, G.; Dabney, W.; Horgan, D.; Piot, B.; Azar, M.; and Silver, D. 2018. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-second AAAI conference on artificial intelligence*.

Lange, S.; Gabel, T.; and Riedmiller, M. 2012. Batch reinforcement learning. In *Reinforcement learning*, 45–73. Springer.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *nature* 518(7540): 529–533.

Schaul, T.; Quan, J.; Antonoglou, I.; and Silver, D. 2015. Prioritized experience replay. *arXiv preprint arXiv:1511.05952* .

Sutton, R. S. 1988. Learning to predict by the methods of temporal differences. *Machine learning* 3(1): 9–44.

Wang, Z.; Schaul, T.; Hessel, M.; Hasselt, H.; Lanctot, M.; and Freitas, N. 2016. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, 1995–2003. PMLR.

Watkins, C. J.; and Dayan, P. 1992. Q-learning. *Machine learning* 8(3): 279–292.