Report template for the project in the course DD2380 at KTH

GROUP 7

Nino Segala Tobias Höppe

08.12.98 27.12.1996

segala@kth.se thoppe@kth.se

Abstract

Reinforcement learning was able to achieve astonishing results controlling agents in computer games in the last years [6]. But an earlier approach for AI in games, so-called $\alpha - \beta$ search is still able to compete. In this paper, we will present a deep Q-learning algorithm and an $\alpha-\beta$ tree search to control agents in UC Berkley' Pac-man Capture the Flag. We will compare both methods and finally test the $\alpha - \beta$ tree search in a competition against other student groups, who used a wide variety of methods for this game. Given the results we get, we can draw a conclusion on the two methods and propose further improvements.

1 Introduction

AI researchers often use simulated environments, such as games, to test new algorithms and methods. In this paper, we will implement a Reinforcement Learning algorithm and $\alpha - \beta$ tree search for UC Berkley' Pac-man Capture the Flag. The game is an alteration of the famous arcade game Pac-man.
First, we will give a short introduction to the rules and dynamics of the game. Two teams control two agents respectively and have to try to bring home as much food from the opponents' side as possible. Once an agent visits the opponents' side, it turns into a Pac-man and is able to collect food while the agent which is in its home base is a "Ghost" and can capture the opponent Pac-man and stop it from collecting food. Once a Pac-man reaches the home base again, the amount of food collected will be counted as score towards the team. If a Pac-man agent would be caught by the ghost on the opponents' side, it will lose all the food which it was carrying and re-spawn in its home base. As an extra, on each side are some power capsules that can only be eaten by Pac-mans and give an agent the ability to catch a ghost instead of the other way around.

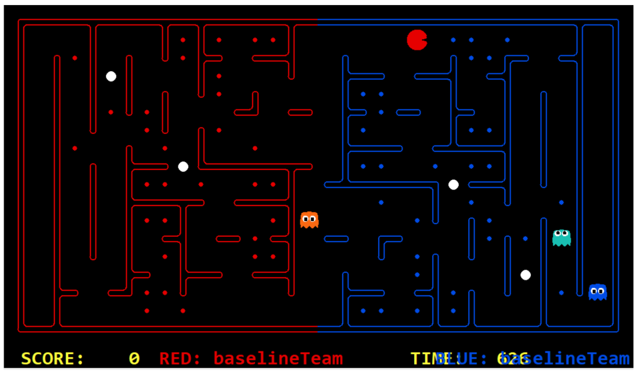

Figure 1: The Pac-man capture the flag environment. The colours represent the sides of each team and the coloured dots food. Also, we can see the power capsules as bigger white dots.

As mentioned we will present a Deep Q-learning algorithm [6] to control the agents. The initial idea was to initialize each team with a Convolutional Neural Network and train both of them by playing against each other as proposed in [2], but training the agents this way turned out to be very difficult. Therefore, we decided to use a simple randomized baseline team as the behaviour policy.
To compete with a working solution we decided to implement an $\alpha - \beta$ algorithm, which is an improved version of the Minimax algorithm. It uses a

search tree where all the possible moves are simulated for a given depth, then each leaf, i.e. the board obtained after the simulation, is evaluated with a heuristic, and finally, we choose the move that returns the best move for us, given that the opponent plays perfectly. With $\alpha - \beta$ some branches of the tree can be pruned, i.e. it is unnecessary to explore these branches, leading to a gain of time which can be used to explore deeper.

1.1 Contribution

We do not necessarily present new ideas on how to tackle the problem. But we give two different approaches which are compared and can build an interesting basis for reproducing and testing methods of tree search or Reinforcement learning. Also, we can give an idea on how to train a Deep Q-learning algorithm if one faces the problem of sparse rewards and a very well working score heuristic for $\alpha - \beta$ search.

1.2 Outline

After we have introduced some related work to our Deep Q-learning approach (Section 2), we will explain our methods for deep Q-learning and $\alpha - \beta$ search in more detail in Section 3. In Section 4 we will explain and analyse our conducted experiments and finally draw a conclusion and discuss some further improvements in Section 5.

2 Related work

Using Neural Networks as an evaluation function for Reinforcement learning was successfully first introduced by [8]. Here, a neural network trains itself to be an evaluation function for the game of backgammon by playing against itself and learning from the outcome. In [6] a convolutional neural network, trained with a variant of Q-learning, is proposed whose input is raw pixels and whose output is a value function estimating future rewards. The method was applied to seven Atari 2600 games from the Arcade Learning Environment and outperforms all previous approaches on six of the games and surpasses a human expert on three of them. In most use cases the complexity of the agents' behaviour is closely related to the complexity of the environment. In [2] it is shown, that this constraint can be overcome by training two deep Reinforcement learning algorithms against each other. This can be very helpful since many simulation environments and games are rather simple. However, the training will be more complex, as according to [2],

simply training the most recent versions against each other can yield in one version falling back and the other one can therefore win without achieving high complexity. Training several agents, which have to interact with each other is even harder, than training one agent. [7] addresses this problem and proposes an algorithm that uses the biased action information of other agents based on a friend-or-foe concept to increase the accumulated performance of the agents when several agents have to act together. Another problem, which many RL applications face, is receiving only sparse rewards. This means that only very few state-action pairs do lead to rewards given a very large state and action space. If this is the case, it is extremely hard to train agents in this environment, as they rarely receive rewards on which they could train. One solution is to design a reward function that can "lead" the agent in the right direction and speed up the learning process [5]. Another choice is to use imitation learning. In this case, the agent does use experience from an "expert". A survey of several methods can be found in [3].
The great von Neumann developed game theory and stated in 1928 the Minimax theorem and the mathematician John Nash defined in 1951 the equilibrium that games with a finite set of actions present. The game trees are used for a long time to search for problems, they are quite intuitive and lead in theory to good results since all the possibilities are tested. But one is very quickly limited by the enormous amount of different games that can be played. Therefore the idea to optimise the searching tree came out. The $\alpha - \beta$ algorithm was developed in parallel by several researchers. The first time the idea was presented, even if the term $\alpha - \beta$ was not used, was in 1955 by John McCarthy in [4]. 3 years later in 1958 A. Newell and A. Simon developed it also on their side in [1]. This algorithm was developed until the 80s, where the optimality of one version of it was shown in 1986 by M. Saks and A. Wigderson.

3 Proposed method

In this section, we will explain in more detail the two approaches implemented to control the Pac-man agent.

3.1 Deep Q-learning

As mentioned, the initial idea was to implement deep Q-learning by letting the agents play again each other and learn. This way was motivated by [2]. However, we experienced a very complicated training process which led us to use a randomized baseline policy as behaviour policy for the agents.

This policy does have one offensive agent, which always goes to the closest food and returns collected food once it is carrying more than 5 pieces. The defensive agent always stays on its team's side and focuses on catching an invasive Pac-man. To introduce some exploration, we do play this policy with a chance of 80% and with a chance of 20% we choose a random action. The history of these policies (state (s), action (a), reward (r), next state (s') was then used to update the parameters θ of the Networks controlling the agents.

To stabilize convergence, we used an extra target Network with parameters ϕ which was updated to the original Networks parameters θ every 100th step. Therefore, given an experience (s, a, r, s'), we updated the parameters of our Network according to the MSE loss

$$\text{loss} = (r + \lambda \max Q_\phi(s, a) - Q_\theta(s, a))^2, \tag{1}$$

where λ is the discount factor, which we set to 0.99, as we the agent is supposed to focus on long term rewards (e.g. winning the game). $Q_\phi(s, a)$ and $Q_\theta(s, a)$ are the Q-values of the state-action pair (s, a) approximated by the Networks with parameters ϕ and θ respectively. Also, to ensure that the samples used for training are not too correlated, we initialized experience replay. That means when receiving a sample (s, a, r, s'), it was stored in a buffer and every update 32 samples were randomly taken from this buffer.

3.1.1 States

We defined the states (e.g the input of the Neural Network) as 4 matrices, which were concatenated when fed to the Network. The first matrix did hold the information about the position of the walls, the second about the position of the food, the third about the position of the agent at play and its partner and the last matrix information about the position of the opponents, if available. We also did experiment with even more information per state, but this made the training time unfeasible.

3.1.2 The Network

As the input had a rather small size of $18 \times 34 \times 4$ we used a rather small Network. It consists of three convolution layers with 16, 32 and 32 channels respectively, a 2×2 max-pool between the first and second layer and two dense layers for. All layer use ReLu as its activation function except the output layer, which has a linear activation function (since we do regression).

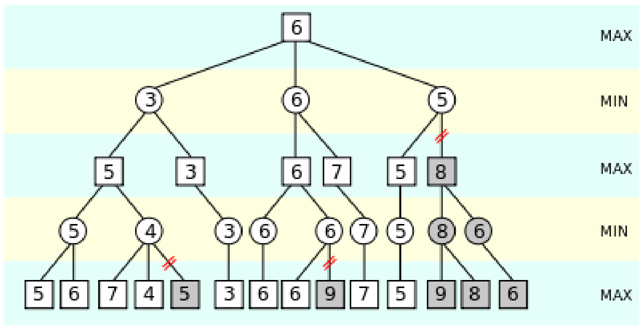

Figure 2: The $\alpha - \beta$ algorithm where the move leading to the best reward we can hope for is chosen

3.2 Alpha-beta

A general example of the $\alpha - \beta$ algorithm is shown in Figure 2. The grey parts of the tree are the pruned branches, which means that we have not looked at them. For example in the right case, the branch below the 5-node can be pruned since we already know that the max-node at the top is equal to the max between 3, 6 and the min-node that contains 5 or below, so we do not need to search the branch below this min-node, because it can not lead to a reward higher than 6.

This algorithm needs to be computed during the game to take each decision. A hard task was to return a move in less than 1 second. It is a very short time letting us not go very deep in the search tree. We first implemented a multiprocessing solution to launch an iterative deepening search (IDS) and stop it just before 1 second to optimise the time. The IDS algorithm computes $\alpha - \beta$ with an increasing depth. The point is to get the best precision being constrained by time. Unfortunately, the multiprocessing thread took 0.5 seconds to close, meaning that only 0.5 seconds were left for the IDS. We decided to use $\alpha - \beta$ with a fixed depth since it was more efficient.
In this game, the positions of the opponents are most of the time unknown. This is problematic in our case since we want to simulate them also. We decided to take the opponents into account when we were able to locate them precisely, i.e. when they were at a Manhattan distance less than 5 from one of our agents of when they had eaten one of our food. In the $\alpha - \beta$ algorithm, we first simulate the main agent, whose turn it was, then we simulate one of the opponents with a priority to the one whose position is known when it is the case, then our agent, after that we simulate the second opponent and then we start again with our main agent. In the case where an opponent's position is unknown, it will skip his turn, meaning that we will simulate the

next player (with the corresponding max-function since the next player belongs to our team), and do not decrease the depth. So if the $\alpha - \beta$ algorithm has a depth n then n players' moves will be simulated. In our case, we fixed the depth at 4, since a depth of 5 led to moves returned after 1 second most of the time. We will discuss further how it has affected our results and how it can be improved.
The design of the heuristic is very important for $\alpha - \beta$ since we decide which move we take based on the evaluations made with the heuristic. We first look if an agent is eaten during the simulation, in this case, we directly evaluate the board with the heuristic. An eaten opponent is highly rewarded whereas if one of our agents is eaten we return a high negative reward. The heuristic is equal to the sum of the following attributes which are normalized and multiplied by a coefficient proportional to its importance for the evaluation of the board:

- The difference between our score and the opponent score
- The collected food by our team
- The collected food by the opponents
- The sum of the distance from each of our agents to the closest food
- The distance between our agents
- The distance from our main agent to our side
- The distance from the teammate agent to our side
- The distance from our main agent to the closest opponent

The distance between our agents is an important attribute to split our agents. Without it, they will most of the time try to catch the same food and do the same. The more closed agents are on the opponent side the more they are penalized, whereas if they are on the column the farthest away from the opponents' side they aren't penalized for it. It allowed our team to split while attacking, without having one agent waiting that its teammate far enough from itself is, to start moving at the beginning. To collect several foods, but returning home early enough to limit the risk of getting caught, we introduced a penalty for the distance from our main agent to our side after it has collected more than 5 foods. So, after reaching this limit, it will tend to come back home, but it will still try to collect food on its way if it is safe.

3.3 Implementation

We first downloaded UC Berkeley's CS 188 Pacman Capture the Flag project on Github and have modified the agent.py file to implement our solution. The game can then be launched from the command-line interpreter where you can specify which team plays with which algorithm.

4 Experimental results

Finally, we will present the results of the competition and analyze them. As the Deep Q-learning algorithm did perform worse than the baseline team it was trained on and lost all simulated games against our $\alpha - \beta$ approach, we decided to use the $\alpha - \beta$ search method for the competition.

4.1 Experimental setup

For the finals, each of the 8 different groups simulated a game of 3 matches against each of the other groups on both the current field (a reference field known by everybody while developing our solutions) and on the random field (a random map we have agreed on chosen specifically for the finals). The results are separated between the current and the random field. The point was to see if there would be some differences in the leader boards or not.

4.2 Analysis of Outcome

Group	11	13	4	8	1	12	**7**	9
Points	103	82	70	67	66	37	**27**	21

Table 1: Leader board on the current field

The outcome can be seen in the Tables 1 and 2. Since our solution is global and not specialised in one field like a reinforcement solution could be if trained only on a specific map, we expected to perform better on the random field than on the current field. And indeed it is what happened: on

Group	11	13	1	8	12	4	**7**	9
Points	97	97	74	72	57	49	**49**	6

Table 2: Leader board on the random field

the current field we finished 7 out of 8 groups with 27 points (7 wins, 29 loses and 6 draws) and on the random field we finished tied on the 6th place with 49 points (16 wins, 25 loses and 1 draws). The leader, group 11, has developed a solution where the opponents' positions are well detected by tracking with the old position, the distances to their agents and their food (where the opponent can't stand since it is not eaten). This was a great advantage for them to have a better overview of the board: where they should defend and where they should not attack. They also had 2 agents with distinct behaviours: one was a defender and one was an attacker. Both agents have a specific destination they should go to depending on the situation like defend this food, catch this food, or avoid the opponent and come back home. It allows them to have more control over their solution and on each behaviour whereas the $\alpha - \beta$ algorithm is supposed to find which goal is the most important given to the situation: if an opponent is trying to catch our agent the heuristic has to return a high score for running away and returning home, whereas in a normal situation it should lead our agent to catch food. One problem is that adding a new feature in the heuristic to make it more complex can override good behaviours that we had before. So adding a new feature takes a lot of time since it first needs to be implemented and then to find the adequate coefficient for this feature in the heuristic we need to watch several games (each game lasts 5 minutes) and change the value of the coefficient until our agents have good behaviours in all situations. Like us, group 11 decided to come back home after having collected 5 food. The second team, group 13, used an approximate Q-learning to control their agents. They had one defender which uses defence features whereas they had a hybrid agent attacker-defender which uses attack and the same defence features. They also used hidden features to determine the goal of the agent. We can observe that there are no huge differences between the two Tables. The only changes are the positions from groups 1, 4 and 12. So all groups came with a solution that was able to play well on a random map.

5 Summary and Conclusions

In conclusion for the Deep Q-learning approach, we can say that more training time was needed. As we were only able to perform about 12 game simulations per hour we had to train for two nights, to even see the first constructive behaviour of the agents. Therefore we conclude that given recent results Deep Reinforcement learning is extremely powerful and outperforms most methods but it needs a lot of engineering and training, which makes it often unfeasible for simpler tasks. Alternatively, one can simplify the state-action space as

Group 1 did. With a smaller and simpler state-action space, one needs a less complex model and less training for convergence. An interesting alternative to this is using $\alpha - \beta$ search as it is easier to implement, debug and more interpretable. The only important engineering which has to be done is the score function. Of course, the drawback of this method is that its complexity is strictly restricted by the depth the search can reach. As we just had one second for searching we couldn't reach deeper than depth four. As far as the engineering of the score function goes, we will propose some improvements in the following section.

5.1 Improvements

Our $\alpha - \beta$ solution could benefit from some improvements. The first one would be to simulate only the agents located in a close neighbourhood of our agent, since an agent far away will not, in most cases, influence the decision of our agent. So it will be possible to simulate more moves of our agent and its close neighbourhood and get therefore better results. Many boards are the same in the simulation tree. So a lot of time could be saved, and thus convert into a deeper search, by avoiding to simulate and evaluate many times the same board. This could be done with a hash function, that calculates a hash number for each board leading to an easy way to compare two boards. We could have a slight difference between the heuristic of our two agents to send one of them at the top of the field and the other one at the bottom. This would, even more, reinforce the separation of our two agents on the field and could resolve some situations where one opponent can block our two agents at the border between the two sides, leading to one free opponent that can collect easily food. It would also be possible to change the heuristic if one agent is blocked when facing an opponent at the border: by going a bit back to create a trap for the opponent or trying to find another way to enter the opponent side. And finally, the heuristic can be made more complex to come back home while being purchase by an opponent in a more optimal way than currently for example (it can still end up in a dead-end sometimes), has better use of the power capsules, try to be caught in a 'good' place if possible since the collected food will be dispersed around, so it is better if the place is accessible than if it is at the end of the map in a dead-end, and symmetrically try to catch the opponents in a 'bad' place.

References

[1] HERBERT A. SIMON ALLEN NEWELL, J. C. SHAW. Elements of a theory of human problem solving. 1958.

[2] Trapit Bansal, Jakub Pachocki, Szymon Sidor, Ilya Sutskever, and Igor Mordatch. Emergent complexity via multi-agent competition, 2018.

[3] Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. Imitation learning: A survey of learning methods. *ACM Comput. Surv.*, 50(2), April 2017.

[4] N. Rochester C.E. Shannon J. McCarthy, M. L. Minsky. A proposal for the dartmouth summer research project on artificial intelligence. 1955.

[5] Laëtitia Matignon, Guillaume Laurent, and Nadine Fort-Piat. Reward function and initial values: Better choices for accelerated goal-directed reinforcement learning. 09 2006.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[7] Heechang Ryu, Hayong Shin, and Jinkyoo Park. Cooperative and competitive biases for multi-agent reinforcement learning, 2021.

[8] Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, March 1995.