The application of PPO and Minimax algorithms in Pacman Capture the Flag game

GROUP 9

Xuecong Liu Carlos Lago Solas

17/11/1997 06/10/1997

xuecongl@kth.se clago@kth.se

Abstract

Games are great environments to develop multi-agent algorithms and put them to the test. The complex dynamics they capture require coordination and cooperation strategies from a good multi-agent algorithm. In this project, we design and implement a Proximal Policy Optimisation (PPO) reinforcement learning algorithm and MiniMax search algorithm to play two agents as a team in Pacman Capture the Flag game. The PPO algorithm was trained on the game states' full information, and the MiniMax algorithm is based on a pre-defined heuristic. These two algorithms displayed their unique advantages and trade-offs in terms of training difficulty and strategic potentials, which would be addressed in detail in this report.

1 Introduction

The study is focused on developing intelligent agents to play Pacman Capture the Flag, where two teams each composed of two agents play against each other in a maze full of walls, food, and power capsules. The objective is to eat the most amount of food from the enemy side, where the agents become Pacman, and return it to our own side, where they become ghosts that can eat enemy Pacmans. By eating power capsules on the enemies' side, the Pacmans become empowered and cannot be eaten by the enemy ghosts for some time. In addition, agents are not able to see the enemies' positions, but rather noisy distances, unless they are close enough. The settings of the game seem simple and fun, but it captures important aspects of multi-agent systems, such as coordination and cooperation.

We tackle this game with two different approaches, reinforcement learning, and search algorithms. For reinforcement learning, we applied Proximal Policy Optimisation (PPO) to control both agents individually, feeding the whole map's information as input to the agent's critic and actor networks for them to evaluate the situation and take actions. The other approach, Minimax, is a tree search algorithm, where possible future game state branches from a current state are expanded and evaluated based on a defined heuristic to select the optimal action.

The main difference between these approaches is that, in PPO, we let the agent networks learn the game rules and develop their own strategies from scratch, whereas Minimax algorithm takes into account the rules from the beginning but is limited to the pre-defined hand-coded heuristic.

Our results are outlined and discussed in sections 2.3 and 3. The Minimax results are not great given the current heuristic. The performance of the PPO agents is sensitive to the training setups, and training with the end game score as the only reward surprisingly turned out to be the best of all and showed strategic potentials.

1.1 Contribution

The main contribution of the project is the successful implementation of PPO in a multi-agent scenario with complex game rules. The impact of different training setups on the performance of the PPO agents is also studied. With proper setup, the PPO agents showed their potential in high-level game strategies with only 2000 training epochs.

1.2 Outline

The study is divided into two parts, one for each applied methods. Section 2 is on Minimax and Section 3 about PPO. Each method is divided into related work and proposed methods, where we cover relevant previous studies in the field and our implementations are explained in detail. The results obtained with the Minimax approach are in Section 2.3. On the PPO method, Section 3.3 goes over the training setup, and Section 3.4 analyses the decision process of the agents. Finally, Section 4 provides a summary and conclusion of the project, including possible future improvements.

2 Search with Minimax

This section will cover in detail our implementation of search algorithms, particularly Minimax with alpha-beta pruning, in the settings of Pacman Capture the Flag.

2.1 Related work

Tree search methods have been extensively using for game-solving. As an example, the study by Marsland [1] reviews the application of Minimax combined with pruning methods such as alpha-beta for playing chess, showing that pruning methods are essential to boost the performance of Minimax by allowing a deeper search.

Another search method is Monte Carlo Tree Search, which does not require using a heuristic, as it simulates games from a state until the end. It has been successfully applied to Pacman, in a study by Pepels et al. [2], achieving first place in a competition. The main drawback of this method is the computation time to run all the necessary simulations, which makes it very difficult to apply in our games, where agents have only 1 second to compute an action.

2.2 Proposed method

Our method is based on Minimax, combined with alpha-beta pruning and iterative deepening. In this particular game, the main difference from other Minimax implementations is that the enemy state cannot be observed unless they are close enough, so the tree needs to be expanded by generating successors of our own game states.

To implement Minimax, we expand the game state tree from the current state according to the possible legal moves, and then branch the new states

until reaching the maximum depth or the time limit. The final nodes are then evaluated according to a heuristic, and the action with the highest heuristic value is selected. The addition of alpha-beta pruning enables the algorithm to prune branches that will never be selected, as better/worse values already exist in the already explored possibilities.

In addition, iterative deepening is used to enable the exploration of all the actions in a balanced manner, by increasing the maximum search depth by one after each search is completed and saving the best overall move.

We chose two different heuristic functions for the offensive agent and the defensive agent, defined as below:

- Offensive heuristic: the objective is to eat and return the enemies' food while avoiding being eaten by the enemies, which could be facilitated by eating the enemy capsule.

$$\begin{aligned} &\text{food\_carrying} * 3 + \text{num\_returned} * 5 + \text{distance\_enemy} * 0.2 - \\ &(\text{distance\_center} * 0.2 * \text{food\_carrying}) + \text{enemy\_scared\_timer} \\ &- \text{distance\_enemy\_food} * 0.1 \end{aligned} \tag{1}$$

- Defensive heuristic: the objective is to be positioned around the place with the most food, trying to eat the enemy Pacmans.

$$\begin{aligned} &\text{enemy\_food\_returned} - \text{being\_pacman} * 10 + \text{remaining\_food} * 2 - \\ &- \text{distance\_center} - \text{distance\_enemy} * 5 - \text{our\_scared\_timer} \\ &- \text{mean\_distance\_our\_food} * 0.25 \end{aligned} \tag{2}$$

2.3 Experimental results

The results of 3 matches against others teams, both in a default map and a random one, are shown in Table 1.

Group	**Default map**	**Random map**
Group 1	66	74
Group 4	70	49
Group 7	27	49
Group 8	67	72
Group 9	21	6
Group 11	103	97
Group 12	37	57
Group 13	82	97

Table 1: Table of points for all groups

Even though our heuristic method performs well against the baseline team, it has trouble winning against other teams. It has worse results on the random map, which is in part due to a flawed defence heuristic. Especially in maps where there are multiple clusters of food, the defence agent sometimes goes away from the cluster of food the enemy is eating, because it reassesses that the centre of our food is now further from the current enemy position. This could be improved by making the defence agent prioritise the clusters where the enemies just ate food.

3 Reinforcement Learning

This section covers related work in Proximal Policy Gradient, discusses our algorithm's design and implementation, and presents the performance of the PPO agents.

3.1 Related work

Actor-critic algorithms are a class of optimisation algorithms for Markov decision processes proposed by Konda et al, where the critic uses TD learning with function approximation architectures and the actor is updated in an approximate gradient calculated based on the critic's information. [3] The idea, in simple words, is to maximise the expected value of the reward in the environment, which is represented as the sum of the product of the probabilities of trajectories and their expected rewards. The critic part calculates the expected reward of a state, while the actor part optimises the probability distribution of the actions based on the critic's estimation. The actor and critic often use neural networks for function approximation.

Proximal Policy Optimisation (PPO) is a family actor-critic reinforcement learning algorithm, proposed by Schulman et. al. It is designed to solve a few problems reinforcement learning algorithms commonly face - difficulty to implement, tune and train, unstable performance, and poor understanding. It boosts the data efficiency and performance stability by introducing a novel objective function with clipped probability ratios, which limits the update of the actor and critic networks. [4] It attains similarly good performance as Trust Regio Policy Optimisation (TRPO) algorithms[5], which gives a hard constraint on the update, but is easier than the latter to implement and has more architecture flexibility.

PPO algorithms have been wildly used as a reliable reinforcement learning method, most notably by OpenAI in a multi-agent hide-and-seek game. [6] Through multi-agent competition, the agents create unexpected strategies

such as shelter building with movable boxes and using ramps to overcome obstacles.

Even though PPO algorithms require less hyperparameter tuning than other reinforcement learning algorithms, hyperparameters can still affect the learning outcomes to a large degree, especially the scales of input and target functions. Henderson et. al. [7] and Andrychowicz et. al. [8] systematically studied the impact of hyperparameters and suggested appropriate range.

3.2 Proposed method

Now we will go into detail on each of the building blocks of the RL agents - PPO, the game state representations, the reward functions, and the training setup.

As explained before, PPO is composed of two networks, an actor and a critic. The more commonly used architecture is for continuous output scenarios, where the actor network returns a normal distribution, as can be seen in Figure 1. However, in this game the action space is discrete, so the output of the network is a softmax and the distribution is now categorical.

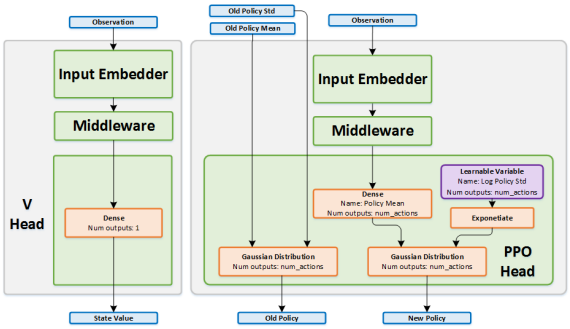

Figure 1: PPO architecture

Both networks take as an input the game state representation, passing then through two dense hidden layers. The structure of the training procedure is as follows:

1. While playing the games, the agents take actions sampled by the output distribution of the actor network. Storing the obtained experiences (current state, action, next state, reward, whether the game terminates).

2. Once enough experiences are collected, the training process starts by

computing the advantage estimation[1] and the probabilities of taking the stored actions with the actor network.

3. The training is performed in mini-batches, where the previous probabilities are compared to the new ones to maximise taking actions that give positive rewards. In addition, the entropy of the action distribution is taken into account for the calculation of the actor loss to favour exploration.

3.3 Training setup

There is a complexity-performance trade-off when it comes to selecting game state representation as input for the PPO networks. Simpler representations require less computation power and result in decent performance fast, but it sacrifices the potentials of reinforcement learning algorithms as the information is not comprehensive. Representations that capture all information of the game state, however, enable the algorithms to figure out higher-level strategies in the long term, but they are painfully computationally expensive. As the point of using PPO in this project is to discover the high-level strategic potentials, we use state representations that have little information loss.

In our state representations, we use the position coordinates of the enemies. One thing to note is that these positions are not given, and the current agent only receives a noisy distance estimation to the enemy if it is far. To improve the estimation, we apply a Kalman filter on all the noisy distance history. We then translate this distance to coordinates by sampling adjacent cells to the last estimated enemy position and picking the cell with the closest distance to the estimated distance. This coordinate is refined by the information of the enemy state (Pacman or ghost), and is estimated with food or capsule positions instead when they eat them.

On top of the state representations, we also experimented with multiple reward setups, and found them critical to the success of this RL method.

The game structure is modified to simulate games with random layouts and starting positions against different opponents.

3.3.1 Training with handcrafted rewards

Game state representation We use a flattened state layout array where wall, food, and capsules are encoded as -1, 1, and 2 respectively. It is then

[1]Difference between the critic network's evaluation of the states and the discounted reward collected after the states (Generalised Advantage Estimation, or GAE).

concatenated with agents' positions, the number of food they are carrying, their scared timers, the game score, and the time left. All digits in the representation are normalised to be on the same scale.

Rewards We considered that it might be too difficult for the agents to learn if there is only the game score as a reward but no rewards during the game (such as reward for eating food and enemy), because the agents might not explore sufficiently to gain enough feedback. Thus, we designed auxiliary rewards to encourage the agents to eat and return food, eat enemies and capsules, protect their own food and capsules, going towards food, and avoid walls. All these rewards, as shown in Table 2, have small scales to make sure the GAE values are between -3 and 3.

Event	Reward
Eating food	±(1/number of food)*0.8
Returning food	±(number returned/number of food)*1.2
Eating capsule	±0.2
Eating player	±0.2
Illegal move	-1.5
Stop	-0.5
Positional reward[1]	$\frac{(PRV-MD)*(37*PRV^2+1975*PRV+242200)}{12650000}$
End game reward	±(1 + gameScore + timeLeft*10)

Table 2: Rewards

[1] PRV - previous minimum distance to food, MD - minimum current distance to food, this reward decreases exponentially so the total reward of reaching a piece of food is not so large compared to the end game reward.

Training The opponents are chosen randomly between a baseline team, a random team, a collection of agents obtained from other teams, and a randomly selected previous version of our PPO agent for self-play. We choose the hyperparameters based on the suggestions in [7] and [8], and reduce the walls in random maps in the first games to make it easier for the agent.

Apart from the random maps, we created special maps for the agent to learn specific tasks, such as eating food or going around walls. Some sample maps are shown in Figures 2 and 3.

Results The agents show a lack of understanding of the game. They go towards the walls even though the action is heavily punished. Before adding a negative reward for the action "stop", the agents tend to choose stop when

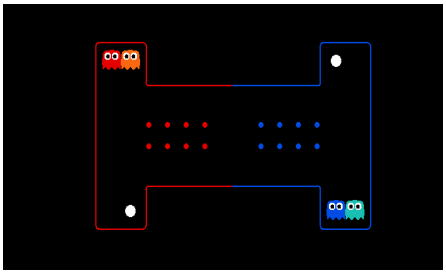

Figure 2: Map designed for learning eating/defending

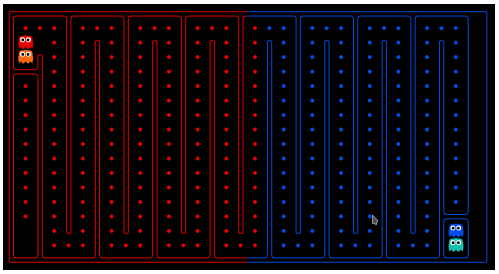

Figure 3: Map designed for learning avoiding walls

facing maps with more walls, despite a large exploration coefficient. This behaviour changed to stubbornly going up and down when stopping is greatly punished. They also seem to not understand their surroundings well, missing the food right next to them sometimes. They failed to beat random agents in simple maps.

3.3.2 Training with game scores as rewards

As the first setup did not work as expected, we tried different configurations after the final competition and found that training with end game score as reward is the key for the agents to understand the game.

Game state representation We experimented with a state representation similar to the previous one, except that the position of the acting agent of the network is centralised in the state array, which causes the state representation to be 4 times larger but provides the networks with an agent-centred view. The previous state representation is also used for comparison.

Reward structure The terminal reward is the end game score, timed with 0.1 for normalisation. All other action rewards are 0.

Training We only use random maps, with random starting positions. Training opponents are a random agent at first, and then include self-play. These settings are to prevent overfitting to certain layouts or opponent behaviours.

Results This reward setting surprisingly yields the best performance, as the agent is not limited by our handcrafted rewards. Even though there is no punishment for stopping or going against the walls, they rarely go against the walls or stop in a corner. Using the agent-centred state representation,

the winning rate against random agents increased stably from 18% to 70%, and the losing rate decreased from 12% to 4% in the span of 35000 games. Without training against the baseline, our losing rate went down from 85% to 75%. The non-agent-centred representation, in comparison, only increased its winning rate to 20% against random agents after 4000 games, by which the winning rate of the agents with centralised training view has reached 40%.

Regarding the behaviour of the agents, we have observed that there is no clear assignation of attack and defence roles between the agents, they flexibly switch between both depending on their position and the enemy ones. In some cases, when an enemy is eating our food, our agents would eat food close to the central border first and then go back to our side to eat the enemy, completing the tasks of returning food and defending at the same time.

3.4 Analysis

Input feature importance is a great value to inspect as they reveal what the agents pay attention to when they make an action. As an example, Figure 4 shows the feature importance for the decision taken by the red agent in Figure 6. The most important feature for the agent in this situation is the enemy position and the food carried by the enemy, highest bars on the right. The other features it pays attention to the most are the information near it, such as walls and food. Detailed feature importance of the game information is shown in Figure 5.

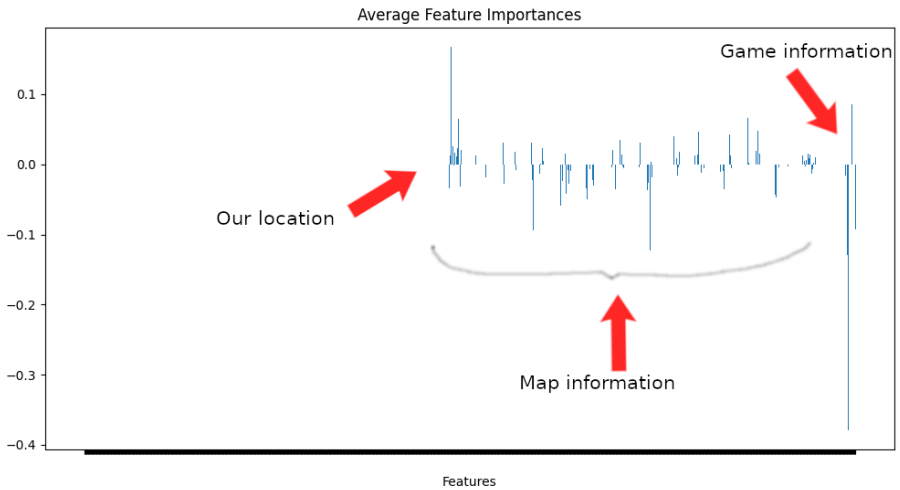

Figure 4: Feature importance (current agent is always located at the centre)

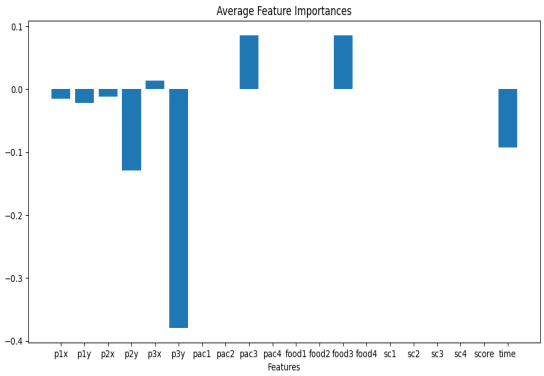

Figure 5: Features importance of the game information

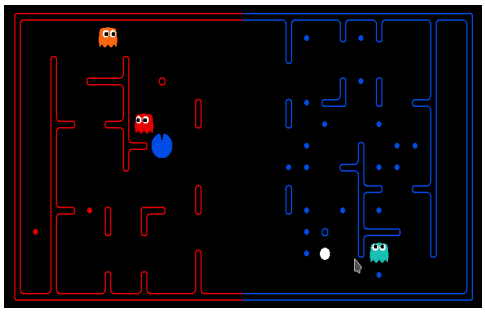

Figure 6: Current game state

4 Summary and Conclusions

In this project, we successfully implemented PPO and Minimax algorithms in a competitive multi-agent Pacman game. Despite the sensitivity to training setup configurations, the PPO agents demonstrated a high-level understanding of the game and its strategies. The Minimax agents also worked well against baseline agents with a pre-defined heuristic.

One of the main findings of this study is that the performance of the PPO agent is significantly better when rewarding just the final score of the game rather than using complex rewards based on specific events. The complicated handcrafted reward only served to hinder the training and let the agents stuck in local optima - a vivid proof against helicopter management. With more training, we have every reason to believe that the agents are going to learn novel strategies of the game.

For future lines, a more comprehensive self-play configuration could be implemented to improve the training efficiency. In addition, Minimax could be combined with RL by using the critic network as its heuristic, reducing computational overhead, enabling it to go deeper, and avoiding the need for a hand-coded heuristic.

References

[1] T Anthony Marsland. A review of game-tree pruning. *ICGA journal*, 9(1):3–19, 1986.

[2] Tom Pepels, Mark H. M. Winands, and Marc Lanctot. Real-time monte carlo tree search in ms pac-man. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(3):245–257, 2014.

[3] Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014. Citeseer, 2000.

[4] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[5] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[6] Bowen Baker, Ingmar Kanitscheider, Todor M. Markov, Yi Wu, Glenn Powell, Bob McGrew, and Igor Mordatch. Emergent tool use from multi-agent autocurricula. *CoRR*, abs/1909.07528, 2019.

[7] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. *CoRR*, abs/1709.06560, 2017.

[8] Marcin Andrychowicz, Anton Raichuk, Piotr Stanczyk, Manu Orsini, Sertan Girgin, Raphaël Marinier, Léonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, Sylvain Gelly, and Olivier Bachem. What matters in on-policy reinforcement learning? A large-scale empirical study. *CoRR*, abs/2006.05990, 2020.