Project Report for Pacman Contest

Zongjian Li
Fudan University
zjli13@fudan.edu.cn

Weijia Chen
Fudan University
chenwj13@fudan.edu.cn

Abstract

In this report, we provide several methods for agents in *Contest: Pacman Capture the Flag*[1] (henceforth referred to as *Pacman Contest*), a simple yet challenging game deriving from the original *Pac-Man* game[2]. written as a course project by University of California, Berkeley.

1 Introduction

In this report, We first analyze some of the key issues for this task, i.e. what makes the project so challenging. Then we propose several versions of the game agents using different algorithms. We first developed three versions of search algorithms – looking for an optimal solution along a large search tree. The algorithms we use are greedy algorithm, expectimax search and Monte Carlo tree search respectively. Then we tried reinforcement learning approaches to solve this problem, adopting Q-learning and the new-fashioned deep Q-learning methods. Finally, we gave game results and success rates for some of the methods we use, as per requirements of the project.

On working with game agents, we adopt the same rules from the original *Pacman Contest*, except that some of our agents can get the exact position of its enemies and allies every time they make an observation. We stick to the map rendered by *layouts/defaultCapture.layout* in all our experiments.

2 Analysis

While *Pac-Man* is an old and classic game with very simple rules, *Pacman Contest* is certainly more complex than its original version. We summarize some key features and difficulties of this project into the following three points:

Partial Observation: Even though our agents are made to fully observe the game state (i.e. able to get exact enemy location) in our later strategies, we start making strategies based on the original game rules – a blurred enemy distance is observed only when an enemy is close enough (within 5 squares in Manhattan distance). Therefore, inferencing enemy's distance is the first thing to do in our project. (See Section 3.1)

Delayed rewards: Different from *Pac-Man* where an agent get immediate reward after eating a dot (get 1 point), or captured by a ghost (lose one life), in *Pacman Contest*, a team does not gain points unless a team agent takes food back from enemy's territory. This poses the challenge of delayed reward. While it does not influence the search algorithm much, the reinforcement learning approach needs to be reconsidered. To solve this problem, we developed our own tricks for rewarding the reinforcement learning agent to act wisely. (See section 4.6)

Multiple agents: As each team has two agents, assigning the role of each agent comes before implementing specific strategies. For some of our methods (e.g. Deep Q-learning agents), we assign specific roles (offensive/defensive) to our agents. Whereas for other methods (e.g. Expectimax search agents), agents are not given specific roles, and can act both offensively and defensively depending on its current state. We believe an optimal agent acts both offensively and defensively, as it best adapts and reacts to different enemy strategies.

[1] http://ai.berkeley.edu/contest
[2] https://en.wikipedia.org/wiki/Pac-Man

3 Search Models

3.1 Position Inference Model

The game does not provide the position of the enemy agent in the distance. Instead, it only offers the Manhattan Distance testing result with a noise. It is thus necessary to infer the position of the enemy agent according to the distance from the noise and make corresponding decisions.

Through the hidden Markov model, the position of the agent is estimated according to the detection results with the noise. The result of the detected noise is an observable variable and the actual position is a hidden variable.

The rules governing the specific action of the enemy agent are unknown. Hence the hidden variable adopts equal-probability transfer, namely the coordinate of every hidden variable x_t has equal probability to move to any available coordinate at $t + 1$ time step.

There are many coordinates where the Agent might be in the game (namely coordinates other than walls). The particle filtering method, which is based on Monte Carlo algorithm, is adopted to accelerate the calculating speed.

Eventually, the particle distribution is changed at every time step. Meanwhile, the number of particles at different coordinates is adopted to estimate the probability of judging the agents coordinate.

In the game, there are two (or more) enemy agents, which should be judged independently. As each teammate of the player (agent it self) implements an action, one observation result is obtained. Compared with a single agent, the combination of all teammates observations yields more evidence variable, which is favorable for improving the inference accuracy. (See *PositionInferenceAgent* under *MySearchTeam.py*)

There is a specific case. When the enemy agent is observable within a short distance (the distance is set to be 5 in the game), all the particles are concentrated in the unit where the enemy agent is.

3.2 Linear Evaluation Function Model

This model evaluates all feasible actions of the current state before each action and selects the action with the best evaluation results.

The evaluation adopts the linear state evaluation function or the linear utility evaluation function. The function of the linear state evaluation is as follows:

$$V(s) = \sum_{i=1}^{n} w_i f_i(s) \tag{1}$$

This function evaluates whether a state is good or bad. In the function, $f_i(s)$ is the value converted from feature i in state s; w_i represents the weight of feature i.

The function of the linear utility evaluation function is listed as follows:

$$U(s, a) = \sum_{i=1}^{n} w_i f_i(s, a) \tag{2}$$

This function offers the rewards obtained by action a in state s. The total reward $U(s, a)$ is the linear sum of several sub-features. In the function, $f_i(s, a)$ is the reward of feature i; w_i is the weight.

Although these two evaluation functions have different meanings and construction means, they have similar means and are thus capable of designing a set of the same interfaces. Both evaluation functions are adopted in different means.

In the process of realizing the model, all feasible actions of a in state s are traversed. After calculating the evaluation function results of action a, the action of the largest result is chosen.

3.3 Expectimax (State Evaluation Or Utility) Model

After the model traverses all possible subsequent states, it selects the action that is most likely to succeed.

The model takes the current state s as the initial state and constantly implements feasible actions and expands the search tree until the game ends.

The basic thinking of the method is to adopt the minimax search algorithm. In the game, every Agent implements an action according to their indices by turns (the preliminary team is random) to conduct the squad fight. Hence the minimax search algorithm is suitable.

The maximum node is the point where the players agent selects the action that is most likely to win. The minimum node is the point where the enemy agent selects the action that makes it most unlikely for the players agent to win.

The game covers a long time interval. Hence searching to final states is unrealistic. It is necessary to Truncate the search in time and replace the

result function with a evaluation function. Within the time limit of only 1 second, this method can only search one or two steps (four agents each take 1 2 steps, in other words the search depth is 1 2). Some improvement is needed. The form of the evaluation function is similar to that in the Section 3.2.

While, when using a utility evaluation function, the returned value of the search function in this model should be changed into the total rewards of subsequent several steps. This is an extension to the minimax search algorithm by us, because the plain version of this algorithm only adopts state evaluation functions.

In fact, the ending state can hardly be searched in the process. The evaluation function is thus important, which influences the strategies and even functions of Agent. (See class *ExpectimaxAgent* and *StateEvaluationAgent* under *MySearchTeam.py*)

3.4 Monte Carlo Tree Search (MCTS) Model

The Monte Carlo Tree Search Model is a non-learning model, which obtains the possibility of winning by carrying out several simulations of the current game state until the ending state. This model selects the action that is most likely to win. We implement Upper Confidence Bound Apply to Tree (UCT) algorithm as our MCTS agent. (See class *MonteCarloTreeSearchAgent* under *MySearchTeam.py*)

4 Reinforcement Learning Methods

Due to the enormous size of the state space, it is basically impossible to design a look-up table for each state-action pair and find the one with the biggest value every time we make an action based on a given state. Therefore, we abandon model-based learning methods which aims at solving Markov Decision Processes. Instead, we evaluate how good each state-action pair is based on agents' interaction with the environment only.

4.1 Q-learning

A Q-learning agent chooses an action in each state to maximize its discounted future rewards, i.e.

$$R(s) = \sum_{t=0}^{n} \gamma^t r_t \tag{3}$$

In Q-learning, the function $Q_\theta(s, a)$, parametrized by θ, is defined as the expected discounted future reward in a state given an action a. The function returns the Q-value of a certain state-action pair. The Q-value can be calculated in various ways, and represents the "quality" of the action a from state s. Whenever the agent takes an action in a given state, the Q-function is updated to match the expected discounted future reward using the function

$$Q_\theta(s_t, a_t) \leftarrow Q_\theta(s_t, a_t) + \alpha * (r_t + \gamma * \max_{a_{t+1}} Q_\theta(s_{t+1}, a_{t+1}) - Q_\theta(s_t, a_t)) \tag{4}$$

where $Q_\theta(s_t, a_t)$ is the Q function, α learning rate, s and a the state and action respectively, and r_{t+1} the reward corresponding with the current state transition. An action selection policy π for a state s can be formed by selecting the action a maximizing Q for all actions possible from s by

$$\pi(s_t) = \arg\max_a Q(s_t, a_t) \tag{5}$$

4.2 Handcrafted Features as Q values

In our first version of Q-learning agent, we approximate Q values with a linear combination of handcrafted features using the following equation.

$$Q(s, a) = w_1 f_1(s, a) + ... + w_n f_n(s, a) \tag{6}$$

The design of features is crucial to this method. Please refer to our code for selection and implmentation of features. Here, we'd like to briefly explain the strategies for our agents.

The defensive agent selects the action that makes it closer to any invading pacman when it is not scared. If it is scared, it tries to move away from the invader. When there's no invading pacman in its own territory, it moves towards the center of the world in its own territory. This is because we believe center is the best place to wait for incoming pacman.

The offensive agent does the following things in order of priority:

- Keep a minimum distance from the enemy ghost
- Eat food
- Eat capsule
- Eat scared ghost

During each training epoch, weight parameters are updated based on each transition

(s_t, a_t, r_t, s_{t+1}):

$$w_i \leftarrow w_i + \alpha * (r_t + \gamma * \max_{a_{t+1}} Q_\theta(s_{t+1}, a_{t+1}) - Q_\theta(s_t, a_t)) f_i(s_t, a_t) \quad (7)$$

Such a training process would require huge amount of computation and take long time. We did not spent time training our Q-learning agent to tune the parameters. Instead, we compare and select features and their weights through human intuition and patient tinkering. (See class *ActionEvaluationAgent* under *MySearchTeam.py*)

4.3 Neural Network as Q function

Recently, inspired by DeepMind's success in combining deep learning with reinforcement learning (Mnih et al., 2015), the AI community has achieved numerous success in applying deep Q-learning to solving different problems. Using convolutional neural network, deep Q-learning method extracts features from each state (represented as stacking of matrices) and output Q-values for different actions. We thus implemented a final version of our agent using deep Q-learning. (See *DQNandBaselineTeam.py*)

4.4 Implementation of Deep Q-learning Agent

Rewards

While Ouderaa (2016) has successfully adopted Deep Q-learning on *Pac-Man* game, the fact that the rewards are delayed in *Pacman Contest* poses a major challenge for the learning agent. Instead of trying to build a highly intelligent agent that learns its ultimate goal of the game – to bring as much food as possible from the enemy domain, we build an agent to achieve different goals in different game status.

Offensive Agent:

- Status 1: In own territory, move towards enemy territory

  While an offensive agent is in its own territory, it should move towards its enemy's territory. Therefore, we give a small negative reward for the agent to move towards enemy's territory, a medium negative reward for moving vertically and a big negative reward for moving against enemy's territory. This reward mechanism successfully "encouraged" the offensive agent to leave its own territory.

- Status 2: In enemy territory, look for food and dodge enemy ghosts

  When the offensive agent has stepped into the other half of the "world", it learns to eat as much food as possible meanwhile tries to dodge the enemy ghosts. A positive reward is given for eating food while a super big negative reward is given for being eaten by a ghost.

Defensive Agent:

- Status 1: Ghost, capture enemy pacman

  A defensive agent only gets a positive reward when it successfully captures a pacman.

- Status 2: Scared ghost, dodge enemy pacman

  If a defensive agent becomes a scared ghost, it will get a medium negative reward for being eaten by pacman.

You may refer to Appendix for detailed information of reward values.

State Representation

The goal of the modeling is to insert as little "oracle" knowledge as possible. This means that the algorithm should learn the environment and rules on its own without explicitly being told how the environment works. Ideally, like humans, an intelligent agent makes decisions based only on frames of the game, with raw pixel values as input vector. Mnih (2015) has proved that reinforcement learning agents learn quite well from image inputs. For *Pacman Contest*, we consider using a slightly different approach, reducing the size of the state space, meanwhile ensuring that all necessary information about the current environment setting is given.

We thus define each state as a stack of matrices. Each matrix represent a key-element to observe. In each matrix, a 0 or 1 respectively express the existence or absence of the element on its corresponding grid position. For instance, a 1 in first row, second column of the wall matrix means that there's a wall in (1,1) position of the grid map.

The state elements for an offensive agent are walls, the agent itself, enemy ghosts, enemy scared ghosts, food and capsules. The state elements for a defensive agent are walls, the enemy pacmans, the agent itself as a ghost, the agent itself as a scared ghost.

You may refer to the appendix for detailed matrix representation of the state.

Network Architecture

With a stack of element matrices as input vector, the neural network outputs Q values for different actions given the current state.

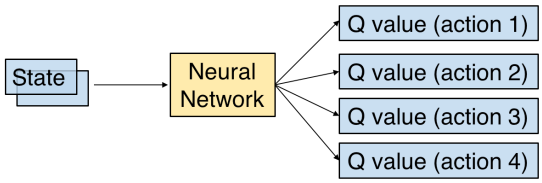

Figure 1: Neural network input and output.

We use the network structure of *DeepQN tensorflow* (Kulkarni, 2012), an optimized neural network structure for playing **ATARI** games written in tensorflow.

The network consists of two convolutional layers followed by two fully connected layers. The convolutional layers are valuable because the game states in the *Pac-Man* game imply the existence of local-connectivity. As a result, CNNs can be extremely useful to improve the classification of features in the game grid.

The first layers use the Rectified Linear Units (ReLu) non-linearity as activation function. The ReLu is defined as $f(x) = \max(0, x)$. Convolutional networks with ReLu activation functions is proven to train several times faster than networks with other activation functions such as tanh (A. Krizhevsky, 2012). The final layer is fully-connected and uses a linear activation function.

Pooling layers are not required due to the fact that the resolution of the input is not excessively large. Moreover, pooling layers would lead to a loss of locational information which presumably is essential for the directional action-selection in *Pac-Man*. Finally, RMSprop is used as an optimization of gradient descent. In RMSprop gradients are divided by the mean squared error of the widths , in order to train more efficiently (T. Tieleman, 2012) .

5 Improvements on Search Models

5.1 Adopt Inherited Relations to Simplify Code Structure

In the whole process, several types of agents (using different models) have been formulated, which are similar in partial functions. Hence the inheritance characteristic of object-oriented programming is adopted to organize such classes of Agents.

The general family of Agents is listed in Figure 2. The functionality of each agent class is introduced briefly in the python source code file, which need no more further description here.

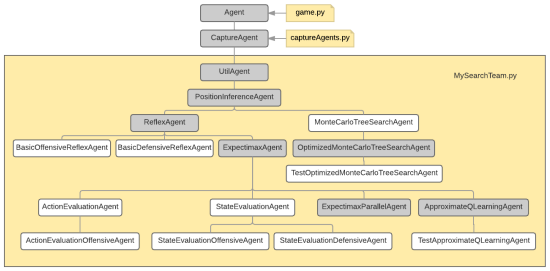

Figure 2: The agent class family. The gray boxes represent the virtual classes and provides some basic functions for sub-classes. The white boxes represent the sub-classes which can be instantiation.

5.2 Adopt Different State Transfer Probabilities for Enemy Agents

In the Inference Position Model, using equal transfer probability, the direction of the enemy can be estimated briefly. However, according to the characteristic of the agents observation & action sequence, this inference may be more accurate.

In two observations, only one enemy agent has adopted an action. Conversely, the other enemy agent stays still. Moreover, it is possible to know which agent stays still. In this way, different transfer probabilities can be adopted to improve the accuracy.

For instance, the agent indices of the playerśteam are (0,2) and the agent indices of the enemy team are (1,3). In agent2's turn, agent0 does not move. Hence the transfer probability of agent0 is $p(action = STOP) = 1$.

We then add a switch to decide whether the $STOP$ transfer action is enabled in Position Inference Model. If we set it to $False$, the inference accuracy improved greatly (shown in Figure 5.2).

5.3 Handle Several Possible Positions Obtained through Position Inference

The position inference has obtained a series of coordinates and their respective probabilities. It is feasible to take the coordinates with the highest possibility as the position of the enemy agent. However, if all inferred coordinates are incorporated in the measurement, more significant results can be obtained.

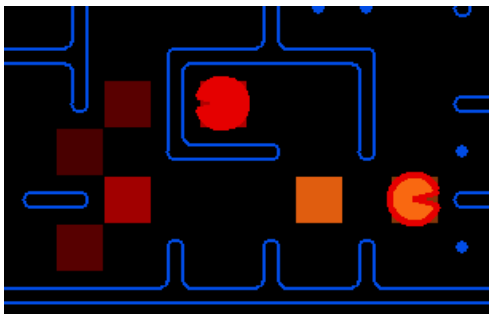

Figure 3: Position inference after optimizations

Our Expectimax Model has expanded the Position Inference Model, allowing it to take full advantage of the multiple coordinates and their probabilities.

Firstly, the inferred coordinate of every agent is arranged in the order of descending probability. The coordinate index of j and its probability of Agent i are represented as $< c_{ij}, p_{ij} >$. A threshold value T is set, restricting each agent to only use the former T inferred coordinates during searching.

Then the combinations of T possible coordinates of all agents respectively implement the minimax search and obtains each searched result and its corresponding action $< v_J, a_J >$. J is the vector of the j value of all agents. In the following, the weighted comprehensive evaluation value V is obtained according to the following formula:

$$V = \frac{\sum_J (v_J \prod_i p_{ij})}{\sum_J \prod_i p_{ij}} \tag{8}$$

Eventually, the action a_J of V_J that is closest to V is selected.

5.4 α-β Pruning

By adding the α-β pruning into the search, the search speed has been greatly improved. As the search depth increases, this improvement becomes increasingly significant.

5.5 Not Considering the Distant Agents

The distant agents exerts small impact on the players judgment. It is necessary to neglect the judgment of agents not within a maze distance (better than use manhattan distance) D to decrease the search workload and further improve the searching speed.

A mechanism has been designed to adjust the D dynamically by max searching depth. In $D = f * d$, d refers to the max searching depth and f is a coefficient. Its default setting is 1.5.

5.6 Iterative Deepening Depth-first Search (IDDFS)

The default time limit of making each action is $1s$. For this reason, how to make full use of the time without exceeding it is the priority of optimization.

In the earliest trial, the search is directly truncated when the time is about to be exceeded. However, different actions have different search depth. The earliest searched action which has the heighest search depth yields a higher evaluation value as the player is about to eat food. Hence the agent will directly choose the first searched action (action $STOP$). The conclusion is that the searching depth of every action should be the same.

The IDDFS meets this requirement. As the searching depth increases, the search will return the former searching results if the search is about to exceeds the time limit.

With this improvement, the agent adjusts the search depth according to the dynamics of workload and makes full use of time.

5.7 Parallelization

Limited by Global Interpreter Lock (GIL) of Python, the multi-thread of Python can only realize concurrency rather than parallelization. Hence multi-process programming is essential to make full use of the multi-core resources on a computer.

The construction of process brings huge costs. It is thus necessary to adopt the process pool to avoid a repeating construction and destruction.

The search tree extends from top to bottom and then aggregates results from bottom to top. The task-based parallel model achieves this demand easily. However, there is no such library in Python. As a result, the parallelization should be realized by traditional means. The difficulty is the upward aggregation of the results returned by each process.

Eventually, the multi-thread is adopted to expand nodes concurrently, which utilizes the characteristics that thread construction has small costs and blocking before returned. We also found out this structure is compatible with the α-β pruning algorithm. By designing the sub-thread as a daemon thread of the parent thread, all subsequent threads can exit together as the parent thread exits.

When the number of threads reaches a threshold (such as CPU number), the threads will halt extension automatically and assign serial search tasks to processes in the process pool. The workload of the serial search can be divided according to the coarse partition or fine partition. The advantage of the coarse partition is the small data transmission and synchronization cost. The advantage of the fine partition is balanced workload. We add a switch to switch between two partition methods.

We find this parallelization improvement works well when the time limit is loose, that is because the data transfer and sychronization cost will be relatively small. Considering the default time limit is $1s$, For the sake of safety, we set the serial expectimax search to be the default super class of agents in the source code.

5.8 Weight of Artificial State Evaluation and Rewards

Although the state evaluation function and the reward function are similar in form, they differ in significance and the designing thinking of features. Although, to some extent, the state evaluation function and reward function can covert to each other.

It is easy to design the features of the reward function. Such as, both eating food and eating the ghost can be taken as a reward. Here it adopts a localized thinking, that makes it is simple to be designed. This kind of agent inclineds towards greed strategies.

The features of the state evaluation function face more restraints. The values of characteristics need to change continuously in continuous actions, which increase the design difficulty yet result in a more holistic agent.

As is discovered in the practice, the game state provided by the game does not include adequate information (such as the number of eaten ghosts), which makes it difficult to design state evaluation functions that meet requirements.

5.9 Methods of Processing Teammates Decisions in Search

At first, the minimum node is adopted to represent a teammates decisions on the search tree. It is found out when teammates get together, none of them will eat the nearby food, because they think the other will do that.

Given considerations to the fact that opportunity nodes will increase the search workload, the minimum node is adopted to represent teammates. Eventually, the above-mentioned problem is solved.

5.10 Optimize MCTS Game Simulation Strategy

Plain Monte Carlo Tree Search Model applies random actions in simulating, which is effective in most chess games. However, it receives a rather low efficiency in this Pacman game. This is because the last action may be "canceled" by the successing action, such as action sequence $< Up, Down >$. The agents do not even move out of their fields. Hence the traditional Monte Carlo Tree Search can hardly provide good performance for this project.

It is thus necessary to adopt one fast and effective action strategy to replace random strategy. We make an interface to load any custom strategy into MCTS with out side effects. However, we find the optimized MCTS is still too slow to simulate enough samples within the time limit. Due to the limitation of the time before submission, we decide to exclude MCTS models from our final contest lineup rather than optimize them (eg. parallelize, truncate before final state), which is a great pity.

6 Results and Analysis

We ran 100 games (1 second time limit) between the following four pairs of agents, and show our results in the Appendix.

- Baseline agents: agents provided by authors (*OffensiveReflexAgent* and *DefensiveReflexAgent* under *baselineTeam.py*. A fully-offensive agnet and a fully-defensive agent.)
- Action-evaluation agents: make decisions based on the values of state-action pairs, i.e. Q-value. (*ActionEvaluationOffensiveAgent* and *ActionEvaluationAgent* under *mySearchTeam.py*. Serial version. Offensive strategy.)
- State-evaluation agents: make decisions based on the values of possible states. (*StateEvaluationOffensiveAgent* and *StateEvaluationDefensiveAgent* under *mySearchTeam.py*. Serial version. Both in charge of offense and defense with inclination.)

- Offensive DQN agent and baseline agent[3]: the offensive DQN agent makes decision based on the output of the neural network. (*OffensiveDQNAgent* and *DefensiveReflexAgent* under *DQNandBaselineTeam.py*)

From the our game result, we come to the following findings:

1. All of our agents beat baseline agents in terms of win rate.

2. Our action-evaluation agents beat all other agents in terms of win rate. It is because this team has two offensive agents, that makes it eat food efficiently. While, this strategy can be easily targeted by opponents' defense agent, such as the fully-defensive agent in baselineTeam.

3. Our state-evaluation agents can't beat action-evaluation agents, but has a higher win-rate over baseline agents. This is a quite interesting phenomenon, reminding us that at this level, the performance of agents depends heavily on their enemies' strategy. These agents are designed to offend and defend, which makes them bring food back in batch. As a result they can get scores gradually. This strategy is safer than those offensive strategy, but may waste action opportunities as they go back and forth, especially when an offensive opponent agent eats a capsules.

4. Our offensive DQN agent didn't perform to our expectation. In fact, it is just slightly better than baseline agents and much worse than the other two pairs of agents. The primary reason is that the agent did not learn to take food back–it only learns to eat food dots and dodge enemy ghost. This easily results in playing hide and seek with enemy ghost in enemy territory. Therefore, the chance of being chased to move onto agent's own territory is low, despite a possible big reward for that action.

[3]It turned out that the defensive DQN agent failed to learn to defend and capture enemy agents. Training a defensive DQN agent is intrinsically harder than training an offensive DQN agent as the chance of bumping into a pacman is far less than bumping into food dots. The agent failed even after we adopted a supervised learning approach to make it learn baseline's strategy. We thus decide to pair our offensive DQN agent with baseline agent. Further investigation is needed to tinker the defensive DQN agent.

We run another 30 games (10 second time limit) between action-evaluation agents and state-evaluation agents. The search depth of the agents increase from 2~5 to 4~9. State-evaluation agents win 11 games, the win rate raise from 31% to 36.7% . That means state-evaluation agents can make better global decision if they see more, while action-evaluation agents do not have this characteristic.

7 Conclusion and Future Work

In this project, we adopted multiple search methods and reinforcement learning methods for agents in *Pacman Contest*. Despite their own strength and weakness, all of them achieve higher winning rates against baseline agents.

We find that even we have severial fine search (in other word, on the basis of calculation) framworks for agents, extracting usful features and setting weights is still a challenge. Weights decides the strategy, we do not know any thing before we give them a try. After days of adjusting weights, we had cognizance of limits of human capacity. We implemented an approximate q-learning virtual class (using Equation 7) to make this procedure automated. Although, it has not yeild results until now. However, we believe that it is a trend that making programs learn by themselves, we are catching this trand.

Even though our DQN agent did not beat all other agents, we believe adopting deep learning techniques is still the right direction to go to work on this problem. Towards the end of the project, we gradually realized that combining deep learning with searching, instead of reinforcement learning, might be a better approach. Basically, *Pacman Contest* is a game of strategy. Like the game of Go, the success of a player requires him to make right decisions consecutively based on a long-run optimal strategy. This makes *Pacman Contest* very different from **ATARI** games, where right moves in consecutive frames could guarantee winning of the game.

However, achieving an AlphaGo-level performance in *Pacman Contest* would require an enormous amount of computational resources, which we could not enjoy at the moment.

As for a further comment, MCTS algorithm is useful nowdays, we hope we can finish our optimization plan to our MCTS agents.

References

I. Sutskever G. E. Hinton A. Krizhevsky. 2012. Advances in neural information processing systems pages 1097–1105.

T. Kulkarni. 2012. Deep q learning in tensorflow for atari. https://github.com/mrkulk/deepQN_tensorflow.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540):529–533.

G. Hinton T. Tieleman. 2012. Neural networks for machine learning. *COURSERA* 4(2).

T. v. Ouderaa. 2016. Deep reinforcement learning in pac-man .

A Game Results

See figure 4

B State representation example

See figure 5, 6 and 7.

C Hyperparameters for deep Q-learning

See tale 1.

D Reward values for different actions

See table 2.

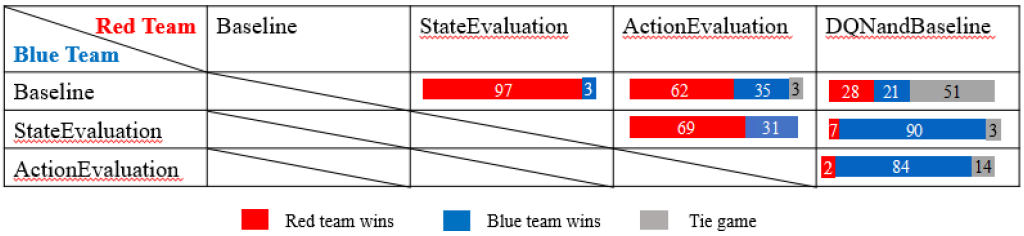

Figure 4: Game Results

Figure 5: Example State

$$\Big(\begin{pmatrix} 1&1&1&1&1\\1&0&0&0&0\\1&0&1&1&1\\1&0&0&0&0\\1&1&1&1&1 \end{pmatrix}, \begin{pmatrix} 0&0&0&0&0\\0&0&0&1&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0 \end{pmatrix}, \begin{pmatrix} 0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&1&0\\0&0&0&0&0 \end{pmatrix}, \begin{pmatrix} 0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0 \end{pmatrix}, \begin{pmatrix} 0&0&0&0&0\\0&1&1&0&0\\0&1&0&0&0\\0&0&0&0&0\\0&0&0&0&0 \end{pmatrix}, \begin{pmatrix} 0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\\0&1&0&0&0\\0&0&0&0&0 \end{pmatrix} \Big)$$

walls *pacman* *ghosts* *scared ghosts* *food dots* *capsules*

Figure 6: State representation for offensive agent

$$\Big(\begin{pmatrix} 1&1&1&1&1\\1&0&0&0&0\\1&0&1&1&1\\1&0&0&0&0\\1&1&1&1&1 \end{pmatrix}, \begin{pmatrix} 0&0&0&0&0\\0&0&0&1&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0 \end{pmatrix}, \begin{pmatrix} 0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&1&0\\0&0&0&0&0 \end{pmatrix}, \begin{pmatrix} 0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0 \end{pmatrix} \Big)$$

walls *pacman* *ghosts* *scared ghosts*

Figure 7: State representation for defensive agent

Hyperparameter	Value
training starts at epoch	5000
minibatch size	32
replay memory size	10000
learning rate	0.0002
discount rate (γ)	0.95
initial exploration	1
final exploration	0.1
steps of exploration	100000
rmsprop epsilon	1e-6
rmsprop decay	0.99

Table 1: Hyperparameters for Deep Q-learning

Agent Role	Action	Reward
Offensive	Eat a food dot	+2
	Eat a capsule	+5
	Eaten by ghost	-500
	Go back to own territory with food	+10*#food-1
	Travel in enemy territory	-1
	Travel horizontally in own territory towards enemy	-1
	Travel horizontally in own territory against enemy	-5
	Travel vertically in own territory	-2
	Stop	-10
Defensive	Travel	-1
	Eat a pacman(ghost)	+100
	Eaten by a pacman(scared ghost)	-100
	Stop	-10

Table 2: Reward Values