Decision Tree and Q-Learning in Pacman

Siqi Chai, Yiran Wu, Zuang Yu, Yinuo Zhang,
Zehua Zeng, Juncheng Pan, Jingyi Zhu, Yisang Luo,
Yibo Zhu, Ji Zhou, Shuo Li

November 26, 2019

Link for Github: https://github.com/ECS171FALL19/Project.git

Abstract

The Ms.Pacman is a classic and challenging video game which provides an instructive model for different machine learning and artificial intelligence algorithms. Our top priority in this project was to win the game safely, and then to maximize the game scores. We used five different methods: Q-learning, approximate Q-learning, Minimax, Alpha-Beta pruning, and Expectimax algorithm to find the optimal solution of Pacman. Based on the comparison of results from these methods, we found that the Approximate Q-learning has the highest winning rate while the decision tree methods generally generated higher average scores.

Introduction

Pacman is one of the most classic arcade games developed in 1980 by Toru Iwatani. Since then, various versions were released, among which the most famous one was Ms. Pacman [1][2]. In the original version, the ghosts' routes were set ahead, making it easy for players to determine strategies to earn the highest score [1][2]. While in the latter version, the behavior of ghosts tended to be random and nondeterministic, which attracted and challenged the players [1].

Pacman and its successors provided great models to study reinforcement learning [3]. To add more challenges to the players, many aspects of the game can be improved further. For example, the agents in the game can evolve while playing, and the ghosts can chase once encountering the agents. Thus, to see if machine learning can be applied to this game and to visualize the training process, we decided to let the game run by itself instead of by a human-controlling console.

Some of the previous studies addressed on such computational intelligence topic and improved the decision of Pacman agent. Lucas et al., for example, developed a tree search algorithm for agents [1]. All possible routes were under consideration, and objects (like edible ghosts, cross-sections on the map, and pills) on the routes weighted differently for the scores that Pacman would earn [1]. In the end, Pacman would choose the route with the highest score. However, since eating edible ghosts had more rewards, sometimes Pacman took risky actions like luring the ghosts to power pills, making the routes chosen not necessarily be the safest one [1][2].

Therefore, we set our priority to win the game (eating all the pills) in addition to maximizing the scores. For our model, we decided to reward "safely winning the game by eating all the pills" the most. This model was expected to maximize scores by winning the game safely with larger benefits instead of consuming the ghosts with short-term benefits. To maximize the winning rate and the

scores, we specified the tree search algorithms alike previous research into Minimax, Alpha-Beta pruning, and Expectimax, and then we applied reinforcement learning to the game.

Methods

1.1 Game setting

We followed the settings of the Ms. Pacman (referred as Pacman below) program written by Berkeley [9] and Starnford [10]. No fruits were included. We included two kinds of maps: smallGrid, which contains two pills and one ghost, and mediumClassic, which was a 20 by 11 layout including the walls with two ghosts. Once Pacman ate powerpills at the corner, ghosts became edible and had more rewards than regular pills. Ghosts could either move randomly or acquire chasing and hiding (once Pacman eat power pills) ability.

1.2 Settings of the reinforcement learning model

Each state included information such as the position of Pacman, the position of ghosts, the position of pills, etc. There were five possible actions here: left, right, up, down and stop. Each game ended when either the Pacman collided with ghosts or it finished eating all pills. Transition function T(s,a,s') represented the likelihood of the agent taking action "a" from state s and ending in state s'. Reward function R(s,a,s') represented the instant reward that the agent took action "a" from the state s and ended in the state s'. We rewarded winning the game with 500 points and losing the game with -500 points. We also rewarded each pill eaten with 1 point and each edible ghost eaten with 200 points. The reward of eating a power pill was the same as eating an ordinary pill. Each step moved cost 1 point as a penalty for time. Discount factor controlled the importance of future rewards in order to help model converge. We preferred the reward that comes early. Alpha was the learning rate that controlled the weight of the current value versus old values when updating.

2.1 Markov Decision Process

There are different types of problems in the real world. Generally, problems are classified into two possible sets, deterministic and non-deterministic. A problem is non-deterministic when its environment is non-deterministic[5]; specifically, given an action and a state, it may result in different next states. In our Pacman game, for example, given an action going up and the position of Pacman, Pacman had some probability of going up, right, left, and down in the next time step. This kind of non-deterministic problems were usually solved based on Markov Decision Process (also known as MDP). We assumed our model followed by MDP because the next state of Pacman only depended on the current state and the action.

If the transition function and the reward function are given in advance, MDP can be easily solved by using policy iteration and value iteration which are called offline planning. However, that was not applicable in our case, because we did not know the transition function and reward function. Instead, we used Q-learning and Approximate Q-learning algorithms which are called online planning.

2.2 Q-learning

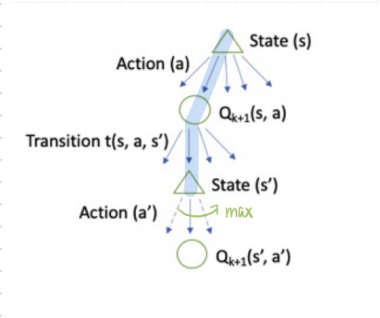


Figure 1: Q-learning algorithm

Q-learning is used in this project as a sample-based, model-free reinforcement learning algorithm (figure 1)[3]. From each state, there are five possible actions, left, right, up, down and stop. The Q-learning model takes actions randomly and explores all possible paths. At each step, the model evaluates how good each action is from each visited state by calculating an associated Q-value. The larger the Q-value is, the better this action from this state is. The Q-value is calculated recursively using the update rule known as q-value iteration[4]:

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')]$$

The Q-value is calculated based on an immediate reward, R(s,a,s'), resulting from the current action and the maximum of a discounted future value.

Each episode the model processes a new sample (s,a,s',r) and the new Q(s,a) value will be updated by incorporating the new sample. By convention, there is a tendency toward keeping the old value. It is done by giving less weight to the new sample[4]:

$$new\ sample = R(s,a,s') + \gamma \max_{a'} Q_k(s',a')$$

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)[new\ sample]$$

2.3 Approximate Q-Learning

The above method, Q-learning, records all Q values in a table. However, in the real world, there are always a large number of states and it may not be possible to learn about every state. Therefore, we generalize the states which have similar features.

Approximate Q-learning, also named Q-learning with function approximation, is to compute a Q value for the state with weighted linear combination of features [6]. The features we use are:
- whether the pill will be eaten
- how far away the next pill is
- whether a ghost collision is imminent
- whether a ghost is one step away

The linear Q-function is:

$$Q(s,a) = w_0 f_0(s,a) + w_1 f_1(s,a) + \cdots + w_n f_n(s,a),$$

where the w_0 is the bias term and n is the number of features[6].

In linear regression, the prediction value $\hat{y} = w_0 f_0(s,a) + w_1 f_1(s,a) + \cdots + w_n f_n(s,a)$, which is Q(s, a) in this case. The target value is $(r + \gamma \max_{a'} Q(s', a'))$; it is the immediate reward plus the best Q value of the next state-action pair. There is a difference between the target value and the prediction value Q(s, a):

$$difference = [r + \gamma \max_{a'} Q(s', a')] - Q(s, a).$$

The square of difference gives the error for the state s:

$$error = \frac{1}{2} \cdot (r + \gamma \max_{a'} Q(s', a') - \sum_{j=1}^{n} w_j f_j(s, a))^2$$

Then differentiate the error with respect to certain weight w_i in order to minimize it[6]:

$$\begin{aligned} \frac{\partial error}{\partial w_i} &= -(r + \gamma \max_{a'} Q(s', a') - \sum_{j=1}^{n} w_j f_j(s, a)) \cdot f_i(s, a) \\ &= -(r + \gamma \max_{a'} Q(s', a') - Q(s, a)) \cdot f_i(s, a) \end{aligned}$$

The rule for updating the weight is [6]:

$$\begin{aligned} w_i &\leftarrow w_i - \frac{\partial error}{\partial w_i} \\ &\leftarrow w_i + \alpha((r + \gamma \max_{a'} Q(s', a')) - Q(s, a)) \cdot f_i(s, a) \end{aligned}$$

3.1 Minimax Algorithm

Minimax algorithm is a dynamic programming algorithm that is designed to find the optimal solutions for multi-player games. For each state that a player reaches, the algorithm returns the maximum score based on all possible actions that other players may choose [13]. In the end, a decision tree is formed, with a depth indicating the number of actions that the top player is going to consider and the numbers on the bottom nodes representing the scores of each action. This is a recursive algorithm works from bottom to top. As opponents, max player always want to maximize the score while the min player will try to minimize it [13]. During our actual implementation, we had a few assumptions with this algorithm: Pacman was the max player, and ghosts were the min players; Pacman moved before the ghosts; Pacman's speed was slightly higher than the ghosts. An example is shown in extended data figure 1. Pseudo code regarding our Pacman game is shown in Extended data pseudo code 3.1.

3.2 Alpha-Beta Pruning

Based on the searching tree constructed from Minimax algorithm, Alpha-Beta Pruning helps reduce the amount of calculations needed for tree traversal when the tree was complex [7]. The pruning process, as the name suggests, skipped the nodes that could not be picked by the Minimax function at the roots in our model [7]. An example is shown in extended data figure 2. Pseudo code regarding our Pacman game is shown in Extended data pseudo code 3.2.

3.3 Expectimax

Similar to Minimax algorithm, Expectimax algorithm is a variant of the classic Minimax algorithm [8]. In our model, the max agents still inherited the max function to select the maximum state

scores from its children. However, the min agents were replaced by average agents which calculated the mean scores of its children instead of looking for the minimum. Such calculations were essential for games involving chances [8]. An example is shown in extended data figure 3. In our Pacman game, we didn't weigh each state score with probability of occurrence because our ghost walked on random paths. Pseudo code regarding our Pacman game is shown in Extended data pseudo code 3.3.

3.4 Evaluation Function

Evaluation function is a heuristic method that is used to evaluate the goodness or performance of current state in a game program [14]. The evaluation is static that it takes in certain features and output a single value. A good evaluation function can raise the rate of winning while doesn't take up much time. In an evaluation function, different features have different weights that will reduce or increase the importance of certain features.

In our Pacman game, we had three evaluation functions in total. Basic Score Evaluation Function was the basic function that was given with the project [9]. It returned the current game score of current state. Game score indicated the overall performances including time used, food eaten, ghost eaten, and whether the game was finished. Game score already included many features, but in a general way, it could not motivate Pacman towards its goal.

4-Feature Evaluation Function took 4 features into consideration. This first feature was game score. We gave this feature a high weight as the basic value of our evaluation score.
Evaluation score = gamerscore * w1

The second feature was the distance to the nearest food. We didn't want Pacman to wonder around. Instead, we wanted it to go directly to the nearest food. Since the bigger the distance, the worse the situation, we penalized Pacman with a large distance. So we subtracted the distance * weight from the evaluation score.
Evaluation score -= DisToNearestFood * w2

The third feature was the distance to the nearest ghost. The smaller the distance was, the worse the situation. So we added this distance to our score. Besides, we wanted to signify this feature when the distance is very small, which indicated that the Pacman was in danger and might lose the game. We then decided to square the distance. For example: if action 1 had a distance of $1^2 = 1$ and action 2 has a distance of $3^2 = 9$, action 2 would get much more scores than action one. Pacman would be more likely to choose the second action.
Evaluation score += (DisToNearestGhost^2) * w3

The fourth feature was the number of food and capsules left. We wanted to minimize this number since all food and capsules should be eaten to win the game. We penalized Pacman with a negative score to motivate it to reduce this number.
Evaluation score -= (NumFoodLeft) * w4

6-Feature Evaluation Function additionally considered two more features to 4-Feature Evaluation Function.

The distance between Pacman and the edible ghost was the first added feature. We added this feature since Pacman could get 200 game scores by eating one edible ghost. Because a score of 200 was significantly higher compared to the cost of one move, -1, we set the coefficient of this feature to be 0.8, which enabled Pacman to eat the nearest edible ghost more intuitively.
Evaluation score += 0.8*min_scared_distance.

The number of legal moves in a state was the second added feature. It was safer for Pacman to move to a state with more legal actions, because more legal actions enabled Pacman to escape from the ghosts in more directions. If Pacman had more directions to escape, it was less likely for Pacman to be caught.
Evaluation score += 0.4*num_legal_moves(state)

4.1 Training and Implementation

For methods from section 2.2 to 2.3, the Pacman explored the map while the game was running. Thus, the setting of Pacman was updated each time. We trained our model with Q-learning and Approximate Q-learning in both the smallGrid and the mediumClassic layout. Specifically, we use grid search with different combinations of learning rates (ranges from 0 to 1) and training episodes.

For the three decision tree methods, each model was played for 500 times before generating winning rate and the average score.

We defined winning as eating all pills without being eaten by a ghost. The winning rates were calculated by dividing the number of times Pacman won with the total number of games played (500). The average score was determined by recording the score each time when the game was finished (Pacman either won the game or was eaten by a ghost).

4.2 Statistical Analysis

To compare different algorithms and parameters, we used built-in functions in R Studio. For instance, to find the relationship between inputs (method, depth, evaluation function) and output (winning rate, average scores), we used model = lm(output features) to build linear models, and used plot(model) and interaction.plot(model) to draw scatter plots and interaction plots. In addition, we used plot() and qnorm() to find Fitted Value Plot and Quantile-Quantile Plot to test assumptions of ANOVA. Finally, we used functions anova (model) and summary (model) to find p-values for each predictor (input) and the correlation between winning rate and average scores.

Result

We implemented our Minimax algorithm, Alpha-Beta algorithm, and Expectimax algorithm based on the Pacman game written by Cindy Jiang for Stanford CS221 course [10]. We implemented our Q-learning algorithm and Approximate Q-learning algorithm based on the Pacman game written by John DeNero and Dan Klein for Berkeley CS188 course [9]. The core codes from these two resources were the same.

1. Performances with algorithms based on decision trees
   We implemented the decision tree algorithms only on mediumClassic map for simplicity.

win rate	Depth	basic eval	4 feature eval	6 feature eval
minimax	2	0.74	0.88	0.91
minimax	3	0.77	0.95	0.93
minimax	4	0.8	0.9	0.94
alpha-beta pruning	2	0.62	0.86	0.9
alpha-beta pruning	3	0.59	0.92	0.93
alpha-beta pruning	4	0.65	0.88	0.92
expectimax	2	0.18	0.92	0.91
expectimax	3	0.41	0.94	0.93
expectimax	4	0.42	0.94	0.94

(a)

average score	Depth	basic eval	4 feature eval	6 feature eval
minimax	2	1275	1539.484	1546.446
minimax	3	1378	1734.632	1707.968
minimax	4	1460	1722	1776.178
alpha-beta pruning	2	1159.008	1528.574	1522.956
alpha-beta pruning	3	1125	1680.795	1711.784
alpha-beta pruning	4	1256.284	1705	1775.458
expectimax	2	-460.456	1561.656	1524.95
expectimax	3	98.51	1687.428	1868
expectimax	4	367.11	1745.518	1739.192

(b)

Figure 2: (a). Winning rate with different algorithms, different depths, and different evaluation functions. (b). Average scores with different algorithms, different depths, and different evaluation functions. Each value in the left three columns was generated after playing the Pacman game five hundred times.

We tested whether the winning rates and the average scores of the Pacman game were affected by different decision tree algorithms, depths or decision functions. As shown in Figure 2, two sets of data were collected, but we were able to show that the values of winning rate and the average scores were correlated with adjusted R-squared equal to 0.95 (Extended data figure 5). We repeated our testing three times for statistical analysis, and then analyzed data with ANOVA (Extended data figure 4).

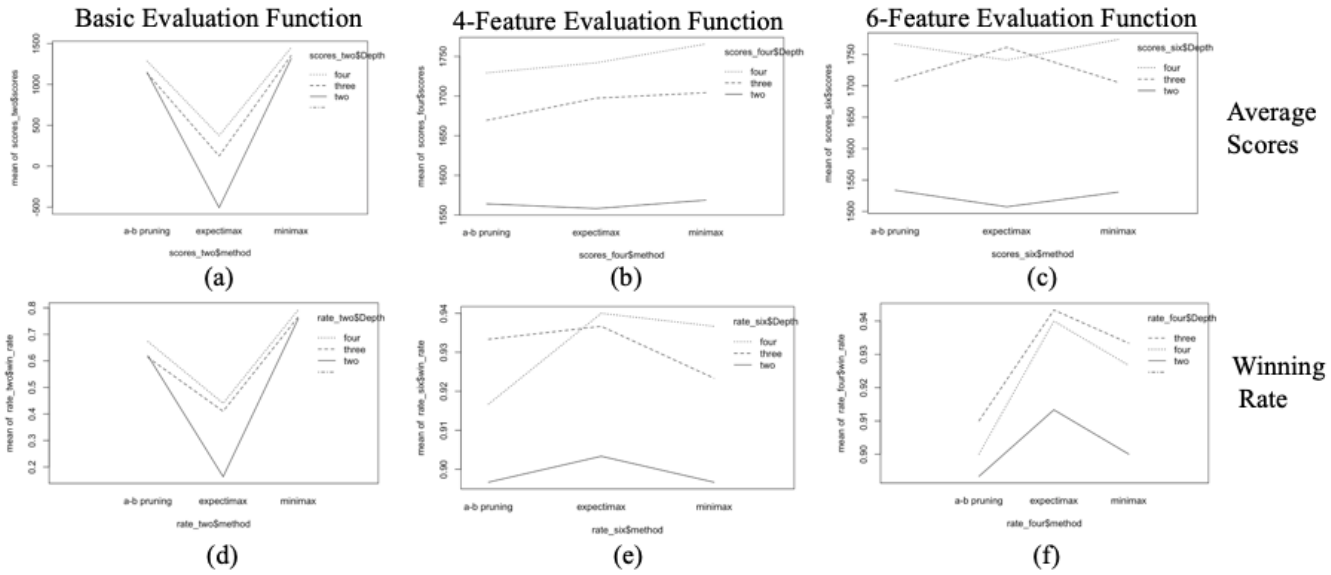


Figure 3: Interaction models. (a)-(c) were drawn on average scores. (d)-(f) were drawn on winning rates.

	p-value (scores)	p-value (winning rate)
method	1.56E-05	1.09E-06
depth	8.93E-02	9.39E-03
evaluation function	2.20E-16	5.16E-16

Figure 4: P-values for scores and winning rate.

Minimax Algorithm is Reliable in Generating High Game Scores

Based on the interactive plots, we found that Minimax algorithm generally produced the highest scores compared to Alpha-Beta pruning and Expectimax algorithms when we applied different features. Expectimax algorithms produced high scores when we used 4-Features and 6-Features Evaluation Function but failed to perform well when we used Basic Score Evaluation Function. Therefore, this algorithm was not reliable (Figure 3. (a)-(c)). Alpha-Beta pruning generally performed well but it acquired smaller average scores than utilizing Minimax algorithm.

Evaluation Functions Significantly Affect both Average Scores and Winning Rate

From the p-values and interactive plots, we observed that when we used Basic Score Evaluation Function, the scores that we obtained from different algorithms and depths were between -500 and 1500 (Figure 3. (a)). However, when we used 4-Feature and 6-Feature evaluation function, average scores fell in between 1500-1700 (Figure 3. (b)-(c)). In addition, when we used Basic Score Evaluation Function, the winning rates were between 0.1 and 0.8 (Figure 3. (d)). However, when we used 4-Feature or 6-Feature evaluation function, winning rates

improved to a range from 0.8 to 0.94 (Figure 3. (e)-(f)). With statistical analysis, the p-values for the evaluation function were both smaller than the threshold of 0.05 when we used winning rate and average scores as outputs (Figure 4). Therefore, the evaluation function significantly affected average scores and winning rates.

Depths Significantly Affect Average Scores but not Winning Rate

Depth indicated the number of steps the Pacman predicts ahead, and we tested its significance based on p-values. The p-value for depth was 0.00939 when the outputs were average scores, which was smaller than threshold of 0.05. Therefore, depth was statistically significant to predict average scores (Figure 4). However, the p-value for depth was 0.0893 when the outputs were winning rates, which was larger than threshold of 0.05 (Figure 4). Thus, depth was not statistically significant to predict winning rates.

2. Performance with algorithms based on Q-learning algorithms.
   We trained our game model with Q-learning and Approximate Q-learning algorithms. These two methods were implemented on the smallGrid and mediumClassic layouts.

   (a) Q-learning

Episodes \ α	0.05	0.1	0.2	0.3	0.4
500	0.60	0.46	0.2	0.54	0.22
1000	0.62	0.72	0.74	1.00	0.72
1500	0.68	0.76	1.00	1.00	0.88
2000	1.00	1.00	1.00	0.9	1.00
2500	1.00	1.00	1.00	0.88	1.00

Figure 5: (left) Winning rate (number of wins over 100 games) of Q-learning algorithm on smallGrid.

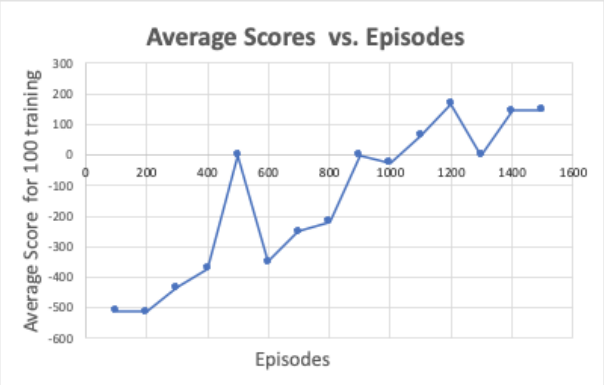


Figure 6: (right) Average Score vs. Episodes, trained 1500 episodes using Q-learning with 0.2 learning rate on the SmallGrid layout.

Based on the table (Figure 5), we found that the Q-learning agent learnt well with 1500 training episodes and 0.2 learning rate for smallGrid. Testing with 100 games, It has a 100% winning rate. We also observed an abnormal pattern when the learning rate was larger than 0.2. We then picked the optimal pair $\alpha = 0.2$ and episodes $= 1500$ and looked at the average scores produced during the training process.

Plotting the average scores for 100 training versus 1500 training episodes, we noticed that there is a growing rate of the average scores with more training episodes (Figure 6.). The average scores started at -500, indicating the Pacman lost the game without gaining any pills. After 1500 training episodes, it ended at around 200, where the Pacman agent not only won the game by eating up all the pills, but also tried to take less steps possible.

We continued experimenting with the Q-learning method with a larger layout – mediumClassic. We tested with learning rate of 0.2 and kept increasing training episodes for a better winning rate. However, the winning rate remained 0 until 9200 training episodes. This process was time-consuming and proved that Q-learning fails to handle a complex problem.

Episodes \ α	0.01	0.05	0.1	0.2	0.25	0.3	0.35	0.5	0.55
20	0.88	0.92	0.94	0.86	0.90	0.96	0.94	0.88	0.94
30	0.90	0.96	0.98	0.90	0.92	0.88	0.92	0.92	0.92
40	0.90	0.88	0.88	0.94	0.98	0.88	0.82	0.96	0.88
50	0.86	0.88	0.88	0.84	0.92	0.90	0.94	0.86	0.90

Figure 7: Winning rate (number of wins over 50 games) of Approximate Q-learning algorithm on the MediumClassic layout.

(b) Approximate Q-learning

The performance of the Approximate Q-learning algorithm was tested on the MediumClassic layout, since we had shown that an easy Q-learning model was able to solve the smallGrid quickly. It is worth noting that just trained with around 40 episodes, the Approximate Q-learning agent has around 90 % winning rate in 50 testing games. We achieved 98% winning rate with both $\alpha = 0.1$, 30 episodes and $\alpha = 0.25$, 40 episodes.

Testing with various learning rates, we noticed that the winning rate was not stable for learning rate larger than 0.3, while it was more reliable for learning rate in the range 0.01 0.25. We then plotted the winning rate versus the number of training episodes to find a pattern among various learning rates.

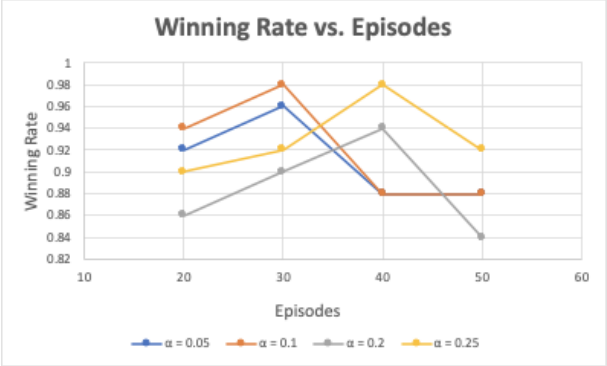


Figure 8: Winning rate for 50 test games verses training episodes.

From Figure 8, we observed a pattern that the winning rate increased with the number of episodes for all learning rates. However, the winning rate decreased when we trained with more episodes because the linear model was overfitted. Generally, the Approximate Q-learning method performed well with just 30 training episodes and we ended up with 1185.34 average scores among 500 test games.

Figure 9: Average score for 500 test games vs. training episodes, trained with $\alpha = 0.25$.

After choosing the right parameters to maximize the winning rate, we plotted the average score for 500 test games over various training episodes. From Figure 9, we observed that there was a growing trend of the average score for 500 test games over training episodes. This indicated that the Approximate Q-learning effectively improved the average score with increasing training episodes.

Discussion

Decision Tree

1. Performance Analysis

   When we ran the games using these decision tree algorithms, sometimes Pacman got stuck at the same position and kept moving back and forth. We believe that such movements occurred because all the decisions at that state generated same scores. Thus, Pacman would start to behave normal again only when the state scores varied hugely (by an approaching ghost, for example).

   One drawback in our decision tree algorithms was that current parameters used were determined manually. As a result,evaluation function is likely to be overfitting. For example, with 6-feature evaluation function and depth of 3, Expectimax algorithm started to outperform Minimax and Alpha-Beta pruning in generating high average scores. However, it failed to outperform with the depth increased to 4. In addition, the performances of certain evaluation function were consistent when the game was repeated 500 times, but the weights of each feature can always be modified to achieve maximum scores and win the game. Therefore, we hope to test more parameters in the future to find the best combinations for decision trees. We can try to apply gradient descent to find the optimal weights specifically.

2. Output Analysis

   We tested on classical algorithms that improved the performances of Pacman game.

   (a) Algorithms

   The three functions basically followed the same logic, which was to use game trees and evaluated future steps to find the next action that would be the optimal. Minimax and Alpha-Beta Pruning had similar winning rate and average score, which was what we expected, since Alpha-Beta Pruning was based on Minimax, but it skipped over unnecessary checks, or prunes the game tree. Indeed, Alpha-Beta Pruning took half time of Minimax to run during our testing (data not shown). What we observed was that Minimax performed slightly better than Alpha-Beta Pruning. We hypothesized that it was because the ghosts took random steps. In our design, we built the Alpha-Beta Pruning algorithm with the assumption that ghosts took the best action against pacman. There were chances that Alpha-Beta Pruning pruned the leaves that evaluated the action the ghost actually took, which letting Pacman make the wrong choice. On the other hand, Minimax went through the whole tree, so it considered all the ghost's actions and enabled a slightly better performance. Expectimax had relatively worse performance on the basic evaluation function compared to other algorithms, but it had much better performances on 4-Feature and 6-Feature Evaluation function. Since the Basic Score Evaluation function only returned state scores and the state scores varied greatly, Expextimax was unreliable by only considering the average of ghost actions.

   (b) Evaluation Function

   For the three decision tree algorithms, all of them didn't have high average scores nor winning rates with Basic Score Evaluation functions. We hypothesized that such phenomenon took place because the default evaluation function returns only the game

score. As detailed in Method 3.4, game score had information of current state but it was too general. With more features included in our evaluation functions, the scores included more information, thus describing the current states more precisely and allowing the Pacman to make choices more accurately. Therefore, both the average scores and the winning rates were improved with a more complex evaluation function. However, adding too many features in evaluation function was more likely to be overfitting. Therefore, it was crucial to find an equilibrium between the evaluation performance and the number of features in evaluation function.

(c) Depth

In the case of different depths, with more steps considered before making a decision, the algorithms could more precisely estimate the action that would maximize the score. Thus, with increased depths, only the average scores were significantly affected instead of the winning rate. In addition, as stated in Performance analysis above, sometimes the pacman stuck at one position. But with more depths, pacman could evaluate more steps ahead, which helped break out of a dilemma quicker. We expect that the average score will be higher with more depths tested in the future.

Reinforcement Learning

From the table (Figure 5), we concluded that the winning rates for Q-learning were generally high. However, since Q-learning tried to explore all possible paths and stores all q-values, it failed to scale complex problems such as mediumClassic which requires exponentially larger training episodes to explore all states.

The Approximate Q-learning algorithm solved the mediumClassic layout with just 30 training episodes and a much faster training speed. It was because Approximate Q-learning algorithm tried to learn some general states and applied on states with similar features. We trained the game during 20, 30, 40, and 50 episodes to find the win rates of testing game, and the result indicated that the winning rates performed better when we have 30 or 40 episodes. We made the assumption that the game was not trained well with 20 episodes, and was overfitted with 50 episodes because the training game was limited to specific examples, and thus reduced the predictive power on the testing game.

One drawback we found when experimenting with Approximate Q-learning algorithms was that the Pacman was not able to distinguish between the edible ghosts and the actual ghosts. We kept our top priority in mind, which is to win the game safely. So, we set the exploration rate, epsilon, relatively small, as 0.05, which let the agents explore unknown paths more carefully. As a result, our Pacman will always run away from the ghosts, which might waste the 50 points bonus if the ghost was edible. We could have experimented with more epsilon values if we have more time and computing power.

We had also coded and tested the Deep Q learning method (Extended Data Concept 1). The Convolutional Neural Network turned out to be too complicated to train. We were aware of the large number of state space so we fed the network with a description of the real frame. The description, a matrix of integers representing agents and game layouts, was much more simple than a frame of pixel. Nevertheless, the CNN was still too hard to train given the large state space. While training on the mediumClassic layout, the score converged from -423 to -200 in the first 100 episodes of training, taking around 10 minutes (Extended Data Figure 8). Eventually, the neural network failed to converge after over 2000 episodes of training. Since we were training on CPU (with keras) only, the computational power might have limited our model. Established research also shows that this method of using state descriptions as CNN input yields satisfactory results after extensive training on GPUs [15].

Comparison between the two approaches

Overall, the Approximate Q-learning algorithm performed better than the decision tree algorithms, because the Approximate Q-learning agent provided generally safer routes and a higher winning rate.

Generally, the Approximate Q-learning agent can be applied on other games directly. However, the decision tree method was designed based on specific games, and therefore hard to transform directly to other problems.

Conclusion

We applied both reinforcement learning and decision tree algorithm to arcade game Pacman for the purpose of maximizing the winning rate and average score. We found that on mediumClassic layout, the maximum winning rate with Approximate Q-learning algorithm (0.98) was higher than that with all three decision tree algorithms regardless of depth and evaluation function parameters (0.95).

Considering which methods generated the highest average scores, we failed to complete the comparison. With limitations of computational power, our Approximate Q-learning model with best learning rate did not converge after 60 training episodes. We observed an increasing trend, but we could not directly compare such tendency with average scores generated from trees. We can improve with Super Computer from online server AWS/Azure so that we spend less time running the data in the future.

References

[1] Lucas, Simon M. "Evolving a Neural Network Location Evaluator to Play Ms. Pac-Man." CIG. 2005.

[2] Bell, Nathaniel, et al. "Ghost direction detection and other innovations for Ms. Pac-Man." Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games. IEEE, 2010.

[3] Bom, Luuk, Ruud Henken, and Marco Wiering. "Reinforcement learning to train Ms. Pac-Man using higher-order action-relative inputs." 2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL). IEEE, 2013.

[4] Nikhil, Sharma. "Introduction to Artificial Intelligence." CS 188 lecture 5 (2019 Fall). PDF file.

[5] Nikhil, Sharma. "Introduction to Artificial Intelligence." CS 188 lecture 4 (2019 Fall). PDF file.

[6] Gnanasekaran, Abeynaya, Jordi Feliu Faba, and Jing An. "Reinforcement Learning in Pacman."

[7] Pearl, Judea. "The solution for the branching factor of the alpha-beta pruning algorithm and its optimality." Communications of the ACM 25.8 (1982): 559-564.

[8] Veness, Joel. "Expectimax Enhancements for Stochastic Game Players." Bachelor thesis, The University of New South Wales School of Computer Science and Engineering, available online: `http://jveness.info/publications/thesis.pdf` [viewed 2011-04-14] (2006).

[9] Berkeley Pacman Code.
`http://ai.berkeley.edu/reinforcement.html` [viewed 2011-04-14] (2006).

[10] Stanford Pacman Code. Jiang, Cindy. "Multi-Agent Pac-Man." Multi-Agent Pac-Man, GitHub, 2019
https://stanford-cs221.github.io/autumn2019/assignments/Pacman/index.html

[11] Qu, Shuhui, Tian Tan, and Zhihao Zheng. "Reinforcement Learning With Deeping Learning in Pacman."

[12] Lague, Sebastian. Algorithms Explained – Minimax and Alpha-Beta Pruning, YouTube, 20 Apr. 2018
https://www.youtube.com/watch?v=l-hh51ncgDI

[13] Wibowo, Haryo Akbarianto. "Create AI for Your Own Board Game From Scratch-Minimax-Part 2." Medium, Towards Data Science, 28 Nov. 2018
https://towardsdatascience.com/create-ai-for-your-own-board-game-from-scratch-minimax-part-2-517e1c1e3362

[14] "Evaluation Function." Wikipedia, Wikimedia Foundation, 19 Nov. 2019
https://en.wikipedia.org/wiki/Evaluation_function

[15] van der Ouderaa, Tycho. "Deep reinforcement learning in pac-man." (2016).

Author contributions

Siqi Chai: Coded ClassicQAgent.py, AppxQAgent.py, DeepQAgent.py based on the pacman project from http://ai.berkeley.edu/. (DeepQAgent.py is inspired by
https://github.com/ele94/deepQLearningPacman,
also referenced in code) Contribution to the method and discussion section of the report. Contribution to the topic decision of the project.

Yiran Wu: Implemented and tested minimax and alpha-beta pruning, implemented evaluation function. Collected most of the data for minimax and alpha-beta pruning. Contributed to minimax, alpha-beta pruning and evaluation function in the paper, including pseudo-code of the two algorithms. Contributed to the discussion sections in the report. Contributed to the command lines of adversarial search in readme.

Zuang Yu: Implemented and tested Expectimax agent function, implemented 6-feature evaluation function. Tested and Collected the data of Expectimax Agent with all of the depth, and some data of Minimax Agent with depth 4. Contributed to Algorithms and Evaluation function in Output Analysis in discussion section in our paper. Contributed to Expectimax Pseudocode in method section. Contributed to command lines in readme.

Yinuo Zhang: Data analysis of the results generated from three decision trees. Contributed to the general organization and design of the report. Contributed to the abstract (partial), introduction (all), methods (minimax, a-b pruning, expectimax, settings, training and implementation), results (basic settings, conclusions for decision trees), discussion (decision tree analysis and overall comparison) and conclusion (partial) sections in the report.

Juncheng Pan: Programmed in R to conduct statistical analysis for minimax, expectimax, and alpha-beta pruning algorithms. Contributed to decision trees and evaluated their characteristics. Researched and reported Deep Q-learning method. Collected and formatted the dataset for learning and other algorithms.

Jingyi Zhu: Trained the Q-learning and Approximate Q-learning models and applied grid search on various parameters. Analyzed and collected testing data from group members and contributed to the method (Q-learning), result (Q-learning and Approximate Q-learning), and discussion (Q-learning and Approximate Q-learning). Contributed to the readme file.

Zehua Zeng: Contributed to Latex editing and formatting. Researched on the Approximate Q-learning algorithm. Trained and tested the game with various parameter values and performed analysis of the data collected. Contributed to the Methods (Approximate Q-learning), Result (Q-learning and Approximate Q-learning), and Discussion (Q-learning and Approximate Q-learning) section of the report.

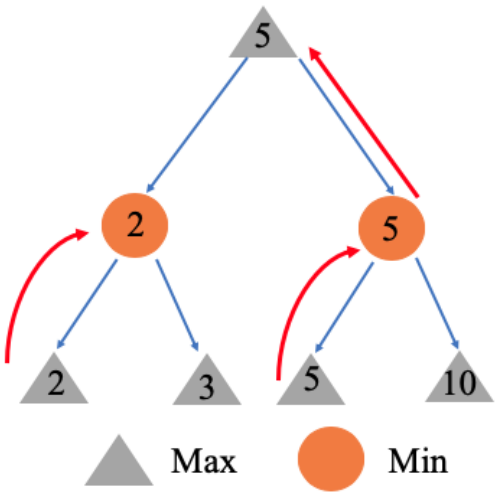
Yisang Luo: Contributed to LaTex editing and formatting. Implemented and applied grid search for both Q-learning and Approximate Q-learning models, then tested and collected data. Contributed to the Methods (MDP) and Result (Q-learning and Approximate Q-learning) section of the paper. Made tables and plots.

Yibo Zhu: Contributed to the Abstract, Conclusion, and References section of the paper.

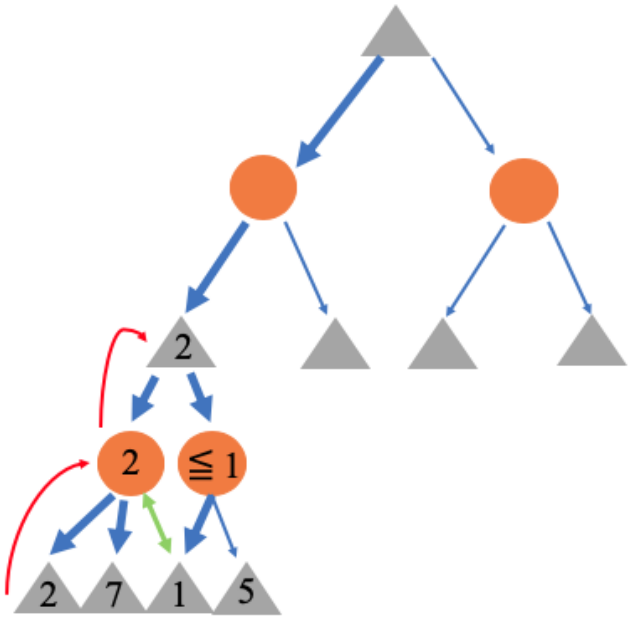
Shuo Li: Freeloader.

Ji Zhou: Freeloader.

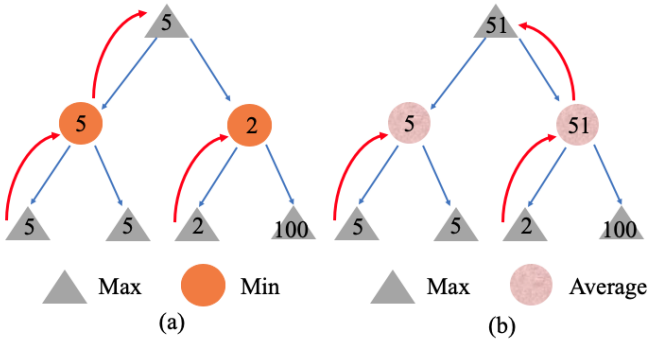
Extended Data (Suppl. mat.)



Extended data figure 1. Example of Minimax Searching Tree with Depth of 2. The min player represented by the circles will always choose the minimum value from its descendent. Once their values are updated, the max player at the top will choose the maximum one, which is 5. By selecting the best result from the worst cases, the player at the top will act to follow the edge on the right first. In this tree, the decision after the min player plays, however, is made by chance. Therefore, after each change of state, a Minimax algorithm needs to be run for the top player for the optimal decision. The depth is the number of actions Pacman would think ahead. By going down the tree, Pacman is evaluating future steps that its opponent and itself can take, and picks the best score that it can get. So a depth of 2 means Pacman would think ahead 2 moves of itself and the ghosts when it needs to decide which direction to go.



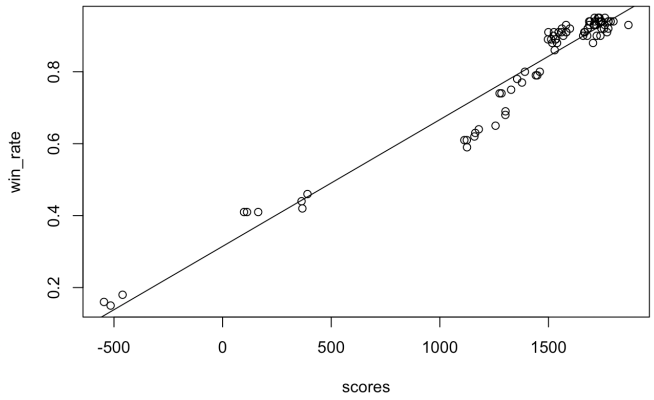
Extended data figure 2. Example of Alpha-Beta Pruning Tree Traversal. Tree traverses from left to right. Triangles have Max function, while circles have Min function. The thickened blue arrows represent the steps go through so far until the first time of skipping. The green arrow represents a situation of skipping. The circles have min function. The tree traverses to 1 right now. Since the circle root always selects the minimum value from its offsprings, the final selected value must be equal to or smaller than 1. With this range in mind, we don't need to look at the leaf of 5 on the right because its parent will never exceed 1. Since the circle root's parent is a triangle node, its left child with value of 2 must be picked when its right child is definitely smaller than 2. Thus, 2 can be updated directly. With pruning, we can decrease the number of leaves that needed to be checked.



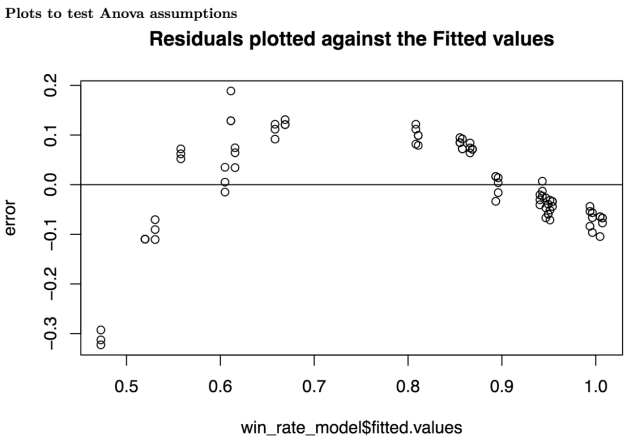
Extended data figure 3. (a) Minimax algorithm with special leaves. (b) Expectimax algorithm. If the decisions are made with minimax algorithm, the top max node will choose the value from its left offspring and decide to follow the left edge (action). However, although the right child doesn't have a large value, its offspring contains a very large value that is "filtered" by the min function. If the top max agent decides to go right, it still has the chance to receive the highest state score of all possible actions. Therefore, by averaging the state scores instead of directly selecting the smallest score, the Expectimax algorithm reflects the average outcome for the max agents to choose.

win rate	Depth	basic eval	4 feature eval	6 feature eval	avarage score	Depth	basic eval	4 feature eval	6 feature eval
minimax1	2	0.74	0.88	0.91	minimax1	2	1275	1539.484	1546.446
minimax2	2	0.74	0.92	0.89	minimax2	2	1283.616	1598.66	1512.246
minimax3	2	0.8	0.9	0.89	minimax3	2	1391.426	1568.262	1533.632
minimax1	3	0.77	0.95	0.93	minimax1	3	1378	1734.632	1707.968
minimax2	3	0.78	0.91	0.94	minimax2	3	1355.874	1664.766	1730.51
minimax3	3	0.75	0.94	0.9	minimax3	3	1328	1713.47	1678.124
minimax1	4	0.8	0.9	0.94	minimax1	4	1460	1722	1776.178
minimax2	4	0.79	0.94	0.93	minimax2	4	1448.594	1775.956	1759.204
minimax3	4	0.79	0.94	0.94	minimax3	4	1441.714	1799.152	1786.106
a-b pruning1	2	0.62	0.86	0.9	a-b pruning1	2	1159.008	1528.574	1522.956
a-b pruning2	2	0.63	0.91	0.91	a-b pruning2	2	1163	1581.994	1560.53
a-b pruning3	2	0.61	0.91	0.88	a-b pruning3	2	1124	1581.78	1517.626
a-b pruning1	3	0.59	0.92	0.93	a-b pruning1	3	1125	1680.795	1711.784
a-b pruning2	3	0.61	0.9	0.95	a-b pruning2	3	1112.724	1659.594	1731.198
a-b pruning3	3	0.64	0.91	0.92	a-b pruning3	3	1179.256	1667.752	1680.48
a-b pruning1	4	0.65	0.88	0.92	a-b pruning1	4	1256.284	1705	1775.458
a-b pruning2	4	0.69	0.92	0.91	a-b pruning2	4	1302.186	1743.826	1769.648
a-b pruning3	4	0.68	0.9	0.92	a-b pruning3	4	1301.666	1739.064	1755.564
expectimax1	2	0.18	0.92	0.91	expectimax1	2	-460.456	1561.656	1524.95
expectimax2	2	0.15	0.93	0.91	expectimax2	2	-515.57	1581.026	1499.476
expectimax3	2	0.16	0.89	0.89	expectimax3	2	-546.086	1531.822	1497.636
expectimax1	3	0.41	0.94	0.93	expectimax1	3	98.51	1687.428	1868
expectimax2	3	0.41	0.94	0.93	expectimax2	3	163.774	1692.37	1688.006
expectimax3	3	0.41	0.95	0.95	expectimax3	3	112.482	1712.146	1728.266
expectimax1	4	0.42	0.94	0.94	expectimax1	4	367.11	1745.518	1739.192
expectimax2	4	0.44	0.93	0.94	expectimax2	4	363.322	1720.642	1741.818
expectimax3	4	0.46	0.95	0.94	expectimax3	4	390.626	1759.918	1742.374

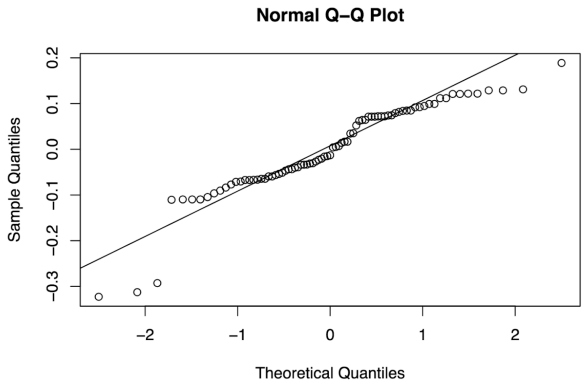
Extended data figure 4. Winning rates and average scores for different features (algorithms, depth, evaluation functions). We repeated each algorithm with different parameters 3 times for statistical analysis.



Extended data figure 5. Scatter plot between average scores and winning rate. (Adjusted R-squared = 0.9494)



Extended Data Figure 6. Fitted Value Plot to test if samples have the same variance



Extended data figure 7. Quantile-Quantile Plot to test if data are normally distributed

Episode	Score
20	-423
40	-418
60	-283
80	-203
100	-152
200	-103
300	-102
400	-101
500	-101

Extended data figure 8. Performances with Training score per episode

Extended data concept 1: Deep Q-learning In Deep Q-learning, it did not extract features like Approximate Q-learning because determining which features were important was subjective. However, it used Convolutional Neural Network as the neural network model. We wanted the network to see what a human-being was seeing when playing the game. The Convolution Neural Network was designed to learn from image data, a set of 3 dimensional matrices with the first two dimensions representing the x, y coordinates of each pixel and the third dimension holding the RGB value. However, given the large state space in the pacman game, we need to compress each frame into a smaller Description Matrix. We represent not only the position of agents in the Description Matrix, but also the direction information. A list of encoding is presented below:

WALL	1
FOOD	2
CAPSULE	3
PACMAN_NORTH	4
PACMAN_SOUTH	5
PACMAN_EAST	6
PACMAN_WEST	7
PACMAN_STOP	8
GHOST_NORTH	9
GHOST_SOUTH	10
GHOST_EAST	11
GHOST_WEST	12
GHOST_STOP	13
SCARED_GHOST_NORTH	14
SCARED_GHOST_SOUTH	15
SCARED_GHOST_EAST	16
SCARED_GHOST_WEST	17
SCARED_GHOST_STOP	18

Extended Data Figure 9. The list of encoding.

In addition, we used a two model structure to assist training. We fed the network to be trained with a list of state Description Matrix as X batch and a list of corresponding predictions based on the Description Matrix given by the other network (Extended data figure 8), as Y batch. After a few training episodes, we updated the model 2 with weights from model 1 using a deep copy method. The model, unfortunately, did not converge well due to the limited computational power.

Extended Data Pseudo code

3.1 Minimax [12]

```
Function minimax(state, depth, agentIndex)
        if agentIndex >= NumAgentIndex
                # Go to the next depth, start with Pacman / agentIndex = 0
                return minmax(state, depth -1, 0)
        # end of iteration, use static evaluation
        if depth == 0 or state in terminate state
                return  evaluationfunction(state)
        if agentIndex == 0 # agent is Pacman
                for each action in state.legalDirections :
                        eval = minimax(state.newState(action, agentIndex),
                                        depth, agentIndex)
                        maxEval = max(eval, maxEval)
                return maxEval
        else # agent is a ghost
                for each action in state.legalDirections :
                        eval = minimax(state.newState(action, agentIndex),
                                        depth, agentIndex)
                        minEval = min(eval, minEval)
                return minEval
```

3.2 Alpha-Beta Pruning [12]

```
Function alphaBetaPruning(state, depth, agentIndex, alpha, beta)
    if agentIndex >= NumAgentIndex
        # Go to the next depth, start with Pacman / agentIndex = 0
        return minmax(state, depth -1, 0, alpha, beta)

    # end of iteration, use static evaluation
    if depth == 0 or state in terminate state
        return  evaluationfunction(state)
    if agentIndex == 0 # agent is Pacman
        for each action in state.legalDirections :
            eval = alphaBetaPruning(state.newState(action, agentIndex),
                                    depth,
                                    agentIndex,
                                    alpha,
                                    beta)
            maxEval = max(eval, maxEval)
            alpha = max(alpha, eval)
            If alpha >= beta
                break
        return maxEval
    else # agent is a ghost
        for each action in state.legalDirections :
            eval = alphaBetaPruning(state.newState(action, agentIndex),
                                    depth,
                                    agentIndex,
                                    alpha,
                                    beta)
            minEval = min(eval, minEval)
            beta  = min(beta, eval)
            If beta <= alpha:
                break
        return minEval
```

.

3.3 Expectimax [12]

```
Function Expectimax(state, agent_index, depth)
        if agent_index >= num_agents
                # Move to next depth starting with Pacman
                return Expectimax(state, 0, depth-1)
        # In terminal state, return current state score
        if depth = 0 or terminal state
                return current state score
        # Pacman's turn, choosing the max state score
        if agent_index ==0
                for each action in all of the legal actions of current agent in current state
                        return max(Expectimax(state.successor(agent_idnex, action),
                                        agent_index+1, depth))
        # ghost's turn, choosing the average of state scores
        else:
                sum_score  = 0
                for each action in all of the legal actions of current agent in current state
                        Eval_score = min(Expectimax(state.successor(agent_idnex, action),
                                        agent_index+1, depth)))
                        sum_score += eval_score
                return sum_score/num_actions
```