CSCI 4998 Final Year Project
SOC 1803
Markov Decision Processes and Reinforcement Learning for Puzzles

1155079423 Tang Sin Yee

April 2019

1 Introduction

Recently, developing artificial intelligence to play games and achieve superhuman performance, like AlphaGo, become more and more popular.

In our project, we aimed to find out game strategy for a game called "1010!". The first part of our project is to apply Markov Decision Process (MDP) with a smaller scale to find an optimal game strategy. The second part, we would apply reinforcement learning, Least Square Policy Iteration(LSPI), on it so that a reasonable game strategy can also be found with a larger scale.

1.1 Research on existed applied game

The first step to start with our project is to look at what is already existed.

1.1.1 2048

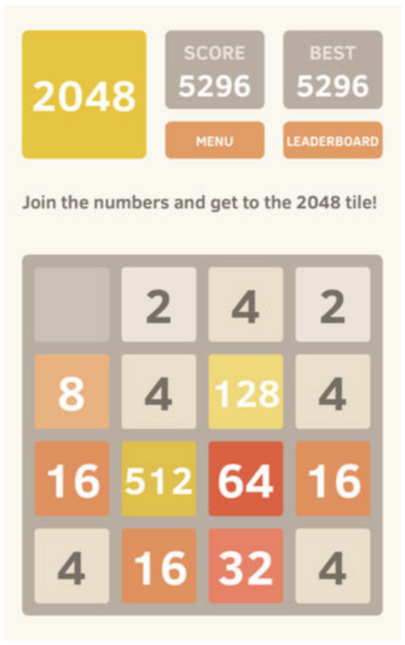


Figure 1: User Interface of 1010!

In ‘The Mathematics of 2048: Optimal Play with Markov Decision Processes’, Markov Decision Processes was applied on a game 2048. Due to the huge number of possible board configurations of 4 x 4 board, it has been simulated with 2 x 2 and 3 x 3 board in the beginning. The strategies and optimal solutions is successfully found out with a smaller board. However, it has obtained the optimal solution of 4 x 4 board up to the “64” tile and failed to find solutions for a higher value of tile because of the huge number of states and long computation time.

1.1.2 Tetris

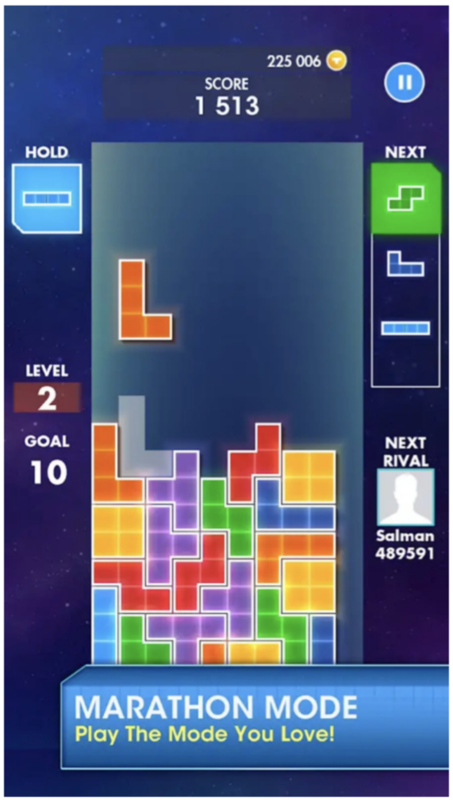


Figure 2: User Interface of 1010!

In ‘Playing Tetris with Deep Reinforcement Learning’, a group of different learning algorithms was applied to playing Tetris. The author claimed that only using a Q function on the game failed to converge, hence, many features were implemented in order to improve convergence. Epsilon-greedy policy, random policy and fitness function were used as the agents of the learning algorithm.

1.2 Project Description

We applied Markov Decision Process (MDP) into the game 1010!, which is a 10×10 board aiming to put pieces on the board, then gain score and cancel lines by completing lines. Markov Decision Process can find out the optimal policy ideally. However, Markov Decision Process fails when there is a huge number of states. With limited memory spaces, we can only find out optimal solution of a narrower version of game, 4x4 board with 7 pieces, by dynamic programming.

In order to enlarge the scale of the game, reinforcement learning is required to solve the MDP problem. Without providing knowledge of the game, such as all states, transition probabilities, rewards, it allows the machine learn from unknown environment.

After studying and making comparison among different reinforcement learning algorithms, like Monte-Carlo (MC), Temporal-Difference(TD) and neural network, we finally decided to apply Least-Square Policy Iteration (LSPI) Algorithm into the game 1010! so as to enlarge the scale up to the original one. The reason of applying LSPI, how it works as well as the performances and advantages/limitations will be discussed later on.

1.3 Objective

We would like to find out game strategy of game 1010! by starting with a smaller scale with MDP and enlarging scale up to the original by LSPI. Since MDP fails to solve by dynamic programming when the scale is too large, we later will use Least-Square Policy Iteration Algorithm, one of a reinforcement learning, to approximate the Q-values. Our aims is to find out game strategy by designing a high quality reward function for MDP and feature function for LSPI and then making comparison of MDP and LSPI.

2 What is 1010! ?

1010! is a puzzle game that is aiming to put as much as possible pieces with different shape onto a 10 x 10 board. When the line is completed, the line will be cleared and score is gained. The board is initially blank. There are 3 random pieces for the user to put on the board. The area of piece that put on the board and the number of lines cleared are contributed to the score. The game will over when there is not enough space for the next piece to put on it.

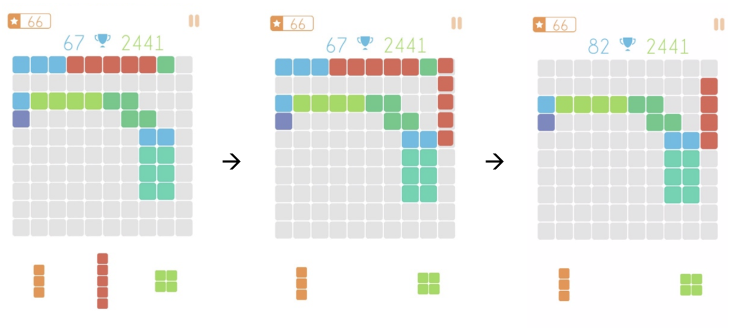


Figure 3: User Interface of 1010!

There are total 19 pieces of different shapes. The shapes with appeared probabilities is shown below.

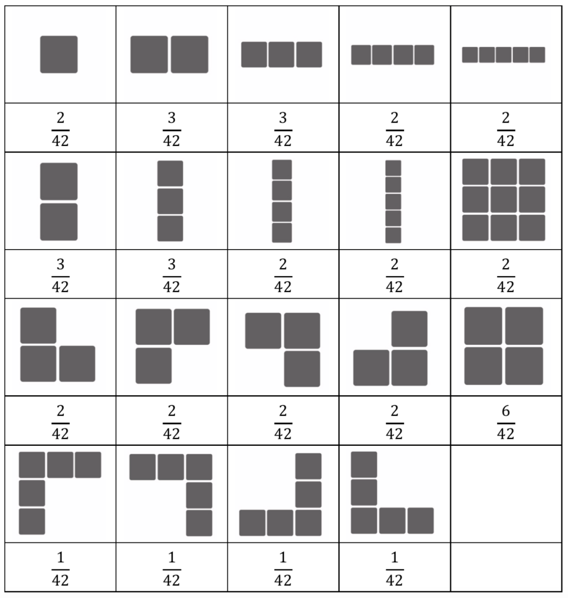

$\frac{2}{42}$ $\frac{3}{42}$ $\frac{3}{42}$ $\frac{2}{42}$ $\frac{2}{42}$
$\frac{3}{42}$ $\frac{3}{42}$ $\frac{2}{42}$ $\frac{2}{42}$ $\frac{2}{42}$
$\frac{2}{42}$ $\frac{2}{42}$ $\frac{2}{42}$ $\frac{2}{42}$ $\frac{6}{42}$
$\frac{1}{42}$ $\frac{1}{42}$ $\frac{1}{42}$ $\frac{1}{42}$

Figure 4: 19 different pieces with appeared probabilities

3 Playing 1010! with greedy algorithms

3.1 Background

At the beginning, we tried to apply greedy algorithm on the game with a smaller size board, 3 x 3, and 3 pieces only to put on. The reason is to find out the differences between the result by greedy algorithm that we have learnt and Markov Decision Process that we are going to learn.

3.2 Greedy Algorithm

We designed three different algorithms

3.2.1 Algorithm 1

Choosing the action that gives the maximum number of line cleared in that state.

3.2.2 Algorithm 2

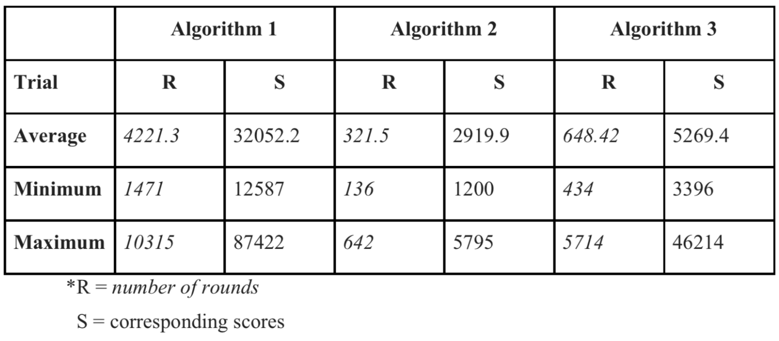
Choosing the action with highest survival probability for next state.

3.2.3 Algorithm 3

The action is randomly chosen by Algorithm 1 and Algorithm 2

3.3 Results

Greedy Algorithm is easy to lose because of lacking in consideration of the future state. I can only consider with the instant state and the next state.



	Algorithm 1		**Algorithm 2**		**Algorithm 3**	
Trial	**R**	**S**	**R**	**S**	**R**	**S**
Average	*4221.3*	32052.2	*321.5*	2919.9	*648.42*	5269.4
Minimum	*1471*	12587	*136*	1200	*434*	3396
Maximum	*10315*	87422	*642*	5795	*5714*	46214

*R = *number of rounds*
S = corresponding scores

Figure 5: Average Number of Rounds Survived and Corresponding Average Scores Using 3 Algorithms

Greedy Algorithm eventually lost the game mainly because it only considers the instant state and the next state. It cannot consider much about the future state.

Using Algorithm 1, it has the largest number round of survival. The major reason of algorithm 1 having the longest survival time is more lines killed more space preserved for that next state, which avoids losing because of lacking in space. However, it only concerns about how to put the block so that this state can have a higher number of line killed. It lacks consideration on the future states. That means if there are several actions made the same number of lines killed, the greedy algorithm will only select one of it without further consideration. (For our coding, the decided action always equal to the earliest one that killed the highest number of lines). Although it can also survive for around 4000 round. The game is finally over.

Using algorithm 2, it tries to maximize the chance for the next state to put the pieces on it. However, this consideration would only affect the result of next round and not the rounds after that, thus, its policy might be good for next round only but not good for the whole game. However, it lacks consideration on kill line. When some pieces put on the board without killing, it is difficult to survive. Therefore, the average around is about 300.

Using algorithm 3, it tries to take action between algorithm 1 and algorithm 2. Although it only improved a little bit comparing with algorithm 2, it still has much lower round comparing with algorithm 1.

As a result, we can see that trying to maximize the number of lines killed can maximize the survival probability at the same time.

4 Playing 1010! with Markov Decision Process (MDP)

MDP can ideally help us to find out optimal policy of the original game. However, solving a MDP problem required dynamic programming. The configuration of the original game is too large that it fails to solve the problem due to performance issue as well as memory issue. Therefore, we decided to narrow down the scale of the game into a board with 3 x 3 and 4 x 4. The number of pieces set to be 3 and 7.

4.1 Equation of Markov Decision Processes

$$V_i(s) = max_{a \in A_s}\{ E[R(s,a)] + \gamma \sum_{s'} \Pr(s') V_{i-1}(s') \}$$

$$\pi(s) = argmax_{a \in A_s}\{ V_i(s) \}$$

$V_i(s)$: Value of State s

s: state

s': future state after taken action $a \in A_s$

i: iterations

$\pi(s)$: policy

γ: discount factor

A_s: all possible action can be taken for each state

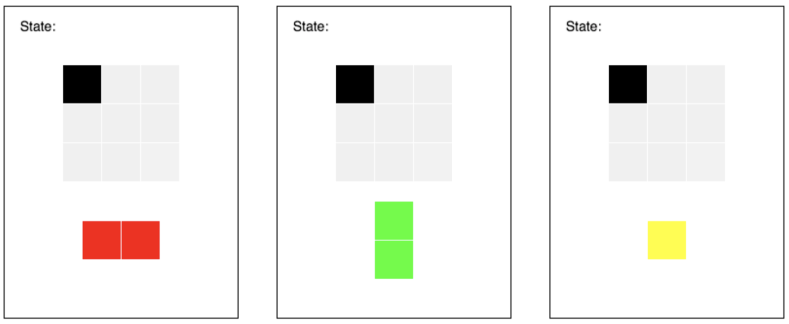
$\Pr(s')$: transitions probability of going to state s' after taking action a

$R(s,a)$: reward function

Figure 6: Equation of Markov Decision Processes

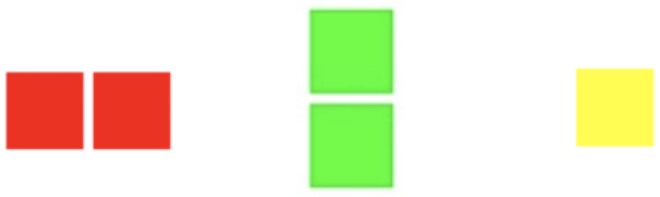
4.1.1 State (s))

A state contains a configuration of a board and a piece. For example in Game1 defined below, a board will form 3 state with 3 difference pieces.



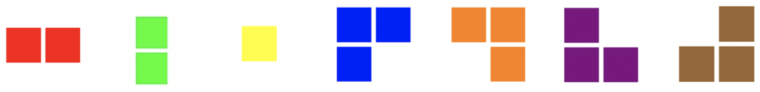
We tried to minimize the game scale of 1010! and increase the difficulty of the game step by step. Below is the Game we defined.

- Game1: a 3 x 3 board with 3 different pieces



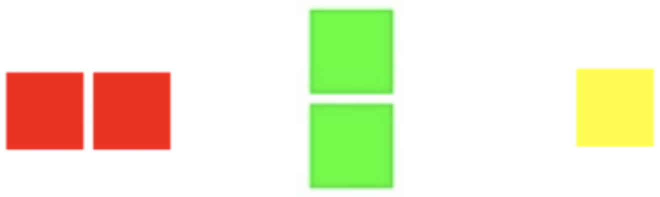
Total number of possible boards = 512
Total number of states = 512 * 3 = 1,536

- Game2:a 3 x 3 board with 7 different pieces



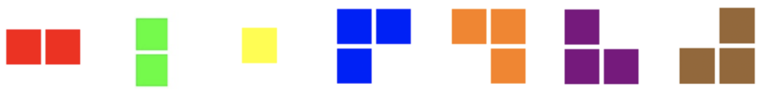
Total number of possible boards = 512
Total number of State = 512 * 7 = 3,584

Game3: a 4 x 4 board with 3 different pieces



Total number of possible boards = 65,536
Total number of State = 65,536* 3 = 196,608

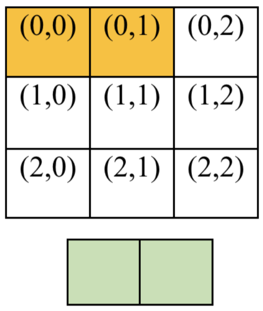
Game4: a 4 x 4 board with 7 different pieces



Total number of possible boards = 65,536
Total number of State = 65,536 * 7 = 458,752

4.1.2 Action (A_s)

(A_s) contain all possible actions of every state s.
For example, the current state s:



A$_s$: $[(1,0),(1,1)],[(1,1),(1,2)],[(2,0),(2,1)],[(2,1),(2,2)]$

A_s contain all possible move of the current state s.

4.1.3 $\pi(s)$: policy

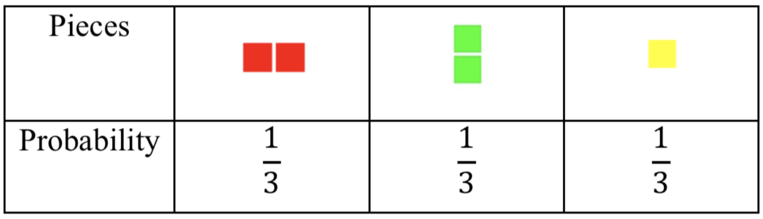
A policy is an action which gains the highest values

4.1.4 Discount factor γ:

The discount factor is to help to adjust the effect of the future states. It ranges between 0 and 1. A higher value means that the value of future states is more important.

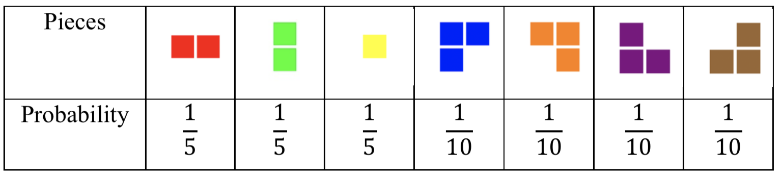
4.1.5 Probability $Pr(s')$

For 3 pieces, we set the probability of getting each piece to be equal



Pieces	🟥🟥	🟩🟩	🟨
Probability	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$

Comparing with the size of the board, a 2 x 2 sized pieces occupy nearly half of the board. If two L-shape pieces generated consecutively, it causes the game over easily. Therefore, we decided to lower the probability of generating L-shaped pieces.

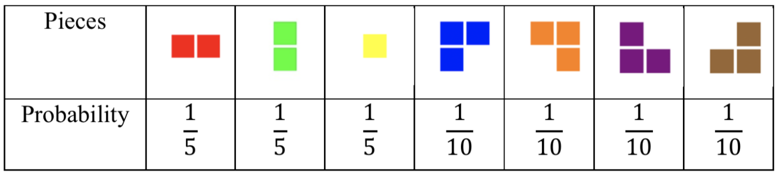


Pieces	🟥🟥	🟩🟩	🟨	🟦🟦🟦	🟧🟧🟧	🟪🟪🟪	🟫🟫🟫
Probability	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{10}$	$\frac{1}{10}$	$\frac{1}{10}$	$\frac{1}{10}$

4.1.6 Reward Function

Reward Function r_1 : (number Of Line Killed)^2 + no. of possible moves of state s'
Reward Function r_2 : (number Of Line Killed)^2
Reward Function r_3 : no. of possible moves of state s'

4.1.7 State Diagram (Example)



Pieces	🟥🟥	🟩🟩	🟨	🟦🟦🟦	🟧🟧🟧	🟪🟪🟪	🟫🟫🟫
Probability	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{10}$	$\frac{1}{10}$	$\frac{1}{10}$	$\frac{1}{10}$

Figure 7: State Diagram of one iteration

This state diagram shows iteration = 20 with $\gamma = 0.8$ and reward function 2
This state has 4 actions (A_1, A_2, A_3, A_4). Each action will go to an intermediate state which is a future board. For each future board, it further goes to the next states according to the transition probability.

For A_1:

$$V_1 = 0 + 0.8 \cdot \left(\frac{1}{3} \cdot 472.39 + \frac{1}{3} \cdot 472.39 + \frac{1}{3} \cdot 475.18\right) = 378.656$$

For A_2:

$$V_2 = 0 + 0.8 \cdot \left(\frac{1}{3} \cdot 472.39 + \frac{1}{3} \cdot 469.97 + \frac{1}{3} \cdot 459.72\right) = 375.754$$

For A_3:

$$V_3 = 0 + 0.8 \cdot \left(\frac{1}{3} \cdot 472.39 + \frac{1}{3} \cdot 469.97 + \frac{1}{3} \cdot 458.04\right) = 373.440$$

For A_4:

$$V_4 = 0 + 0.8 \cdot \left(\frac{1}{3} \cdot 472.39 + \frac{1}{3} \cdot 469.97 + \frac{1}{3} \cdot 458.04\right) = 373.440$$

The highest value is V_1 among the others, as a result, A_1 will be selected as the policy of this states.

5 Optimal solution for MDP

Optimal policy is defined as the policy that achieves the highest value for every state. We have defined the optimal value function as

$$V_i(s) = max_{a \in A_s}\{ E[R(s,a)] + \gamma \sum_{s'} \Pr(s')\, V_{i-1}(s') \}$$

$$\pi(s) = argmax_{a \in A_s}\{ V_i(s) \}$$

Policy and value will be updated by calculating value from this equation repeatedly. The optimal policy is the policy that remains unchanged after converging of value. Since gamma values between 0 and 1, each iteration has a contraction. The values will finally converge and an optimal solution can be found.
In our case, around 20 iterations are needed for convergence. The number of iterations in order to converge an optimal policy is as follow,

	Iterations for converging
Game1	20
Game2	18
Game3	16
Game4	16

6 Evaluation of result

With the obtained optimal solutions, games with three pieces can survive infinite rounds in most of the time, while games with seven pieces can only survive hundreds of rounds. We would like to investigate mainly on the relationship between values of gamma and reward function towards Q-function with respective to the different number of pieces with 4 x 4 board.

6.1 Games with Three Pieces

Given three pieces, both of the games with 3 x 3 and 4 x 4 boards are able to survive infinite rounds in most of the time. Hence, the following collected data were the average scores under the same number of round, which is 10,000.

6.1.1 Values of Discount Factor

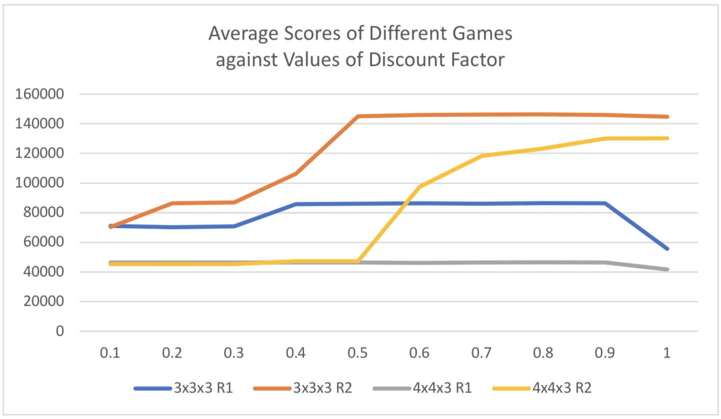


Figure 8: Graph of Average Scores of Different Games against Values of Discount Factor

A significant trend is found in the above graph, scores have surged and stayed stable after upsurge. In 3 x 3 board, an increase is found between values of gamma 0.3 and 0.5. After the increase, the scores remain stable. In 4 x 4 board, an increase is found between values of gamma 0.5 and 0.9. After the increase, the scores maintain stable.

There are a few similarities between 3 x 3 board and 4 x 4 board are found. First, the value of gamma 0.9 will get the highest scores with games of three pieces. Since a larger value of gamma will take future states' values as more important and have a larger influence on future states' values. It is believed that the value of gamma 0.9 would be best to reward functions 1 and 2. Second, there is no remarkable effect on changing values of gamma while applying reward function 1. Besides, there is a significant drop when gamma was 1 while applying reward function 1. The reason will be explained below.

On the other hand, A difference of trend of scores between 3 x 3 board and 4 x 4 board is found. The scores of the game with 4 x 4 board are lower than that with 3 x 3 board under all circumstances.

The reason behind is that it required more steps to complete lines due to the larger dimension while the scores for completing lines were the same, and thus, the scores are lower with the same number of rounds.

6.1.2 Reward Functions

We will mainly focus on the relationship between reward functions and scores because the games with three pieces can survive infinite rounds in most of the time and we can hardly compare the survival time.

Obviously, applying reward function 2 got more scores than applying reward function 1. As stated before, reward function 1 depends partially on the number of line killed and the number of possible moves for state s', while reward function 2 depends only on the number of lines killed. It is believed that the factor of the number of possible moves is not contributed to the score directly. Hence, reward function 1 is not good enough to maximize the score. On the other hand, the score is directly proportional to the number of lines killed and so reward function 2 will get higher scores than reward function 1.

Besides, as mentioned before, there is a significant drop when gamma was 1 while applying reward function 1. A larger value of gamma will lead to less consideration on the scores due to the factor of the number of possible moves.

For games with three pieces, the highest score would be obtained by applying reward function 2 with the value of gamma 0.9. Since games with three pieces can survive infinite rounds in most of the time, we could not conclude any discovery on the impact of values of gamma and reward functions towards the survival time.

6.2 Games with Seven pieces

Given seven pieces, both of the games with 3 x 3 and 4 x 4 boards are able to survive at most hundreds of rounds. Since the data collection of 3 x 3 and 4 x 4 boards are distinct, we will further divide this part into two section, game with seven pieces in 3 x 3 board and game with seven pieces in 4 x 4 board.

The reason of distinct data collected of 3 x 3 and 4 x 4 boards is that the small dimension of 3 x 3 board hardly places the “L-shaped” pieces continuously. Therefore, no matter which value of gamma or reward function, it will easily game over due to its space limitation. Since the limitation of space in 4 x 4 board is lower than that in 3 x 3 board, its performance is better than that in 3 x 3 board and similar to the result obtained in games with three pieces.

6.2.1 Values of Discount Factor of 3 x 3 board

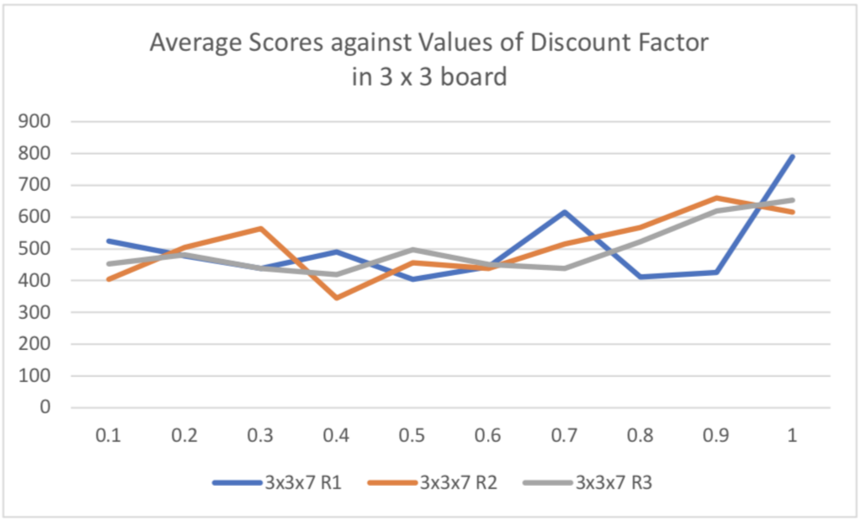


Figure 9: Graph of Average Scores against Values of Discount Factor in 3 x 3 board

From the above graph, scores jumped rapidly between values of gamma 0.9 and 1 and it attained the highest scores under the value of gamma 1. Since a larger value of gamma will take future states' values as more important and have a larger influence on future states' values. Similarly, the scores are the highest with the value of gamma 1. Besides, it is noted that the score of reward function 2 increased under values of gamma 0.3 and decreased instantly under values of gamma 0.4. Apart from that, there is no remarkable trend between values of gamma 0.1 and 0.9, the scores rose and fell without sequence.

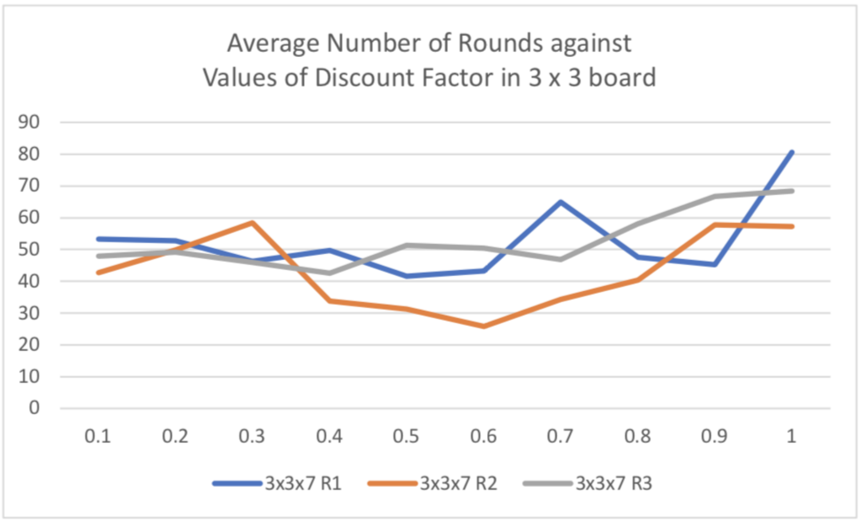


Figure 10: Graph of Average Number of Rounds against Values of Discount Factor in 3 x 3 Board

On the other hand, the trend of the number of rounds is similar to that of scores. The number of rounds is also mainly highest between values of gamma 0.9 and 1 and increased and decreased without sequence between values of gamma 0.1 and 0.9. Similar to the average score, the number of round of reward function 2 increased under values of gamma 0.3 and decreased instantly under values of gamma 0.4. Also, observed from the above result, the score is directly proportional to the number of rounds survived.

From the above two graphs, we found that the trends of scores and number of rounds of three reward functions are similar. There is no big difference between the shapes of the two graphs. Reward function 1 got the highest scores and survived the largest number of round, though the difference between the three of them is small. As a consequence, three reward functions performed similarly and applying reward function 1 with gamma value 1 performs the best. The reason will be explained below.
Different from the discovery above, reward function 1 performs the best in games in 3 x 3 board with three pieces. Also, the score is highest under the value of gamma 1.

6.2.2 Values of Discount Factor on 4 x 4 board

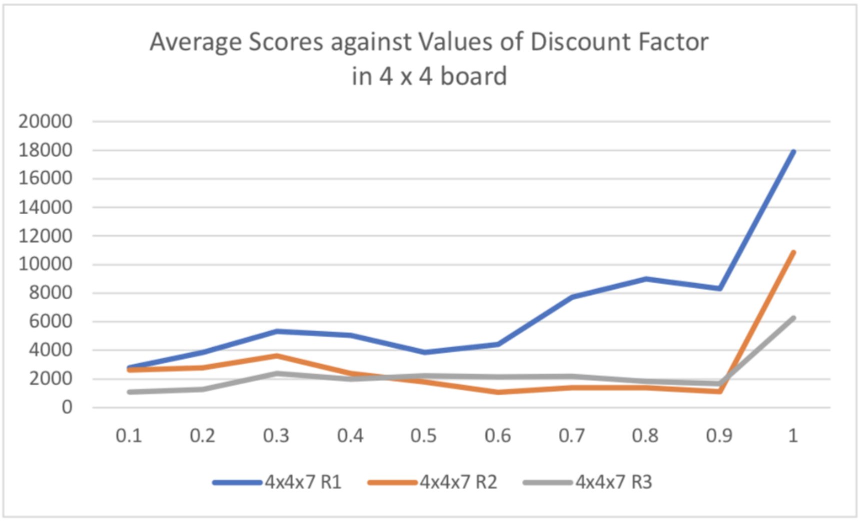


Figure 11: Graph of Average Scores against Values of Discount Factor in 4 x 4 board

As we can see, there is a trend that the scores jump rapidly between values of gamma 0.9 and 1, which is similar to the discovery mentioned before. Besides, it rose between values of gamma 0.1 and 0.3 and fell values of gamma 0.3 and 0.5. Similar to the result in 3 x 3 board with seven pieces, the scores form a little crest when the value of gamma is 0.3. In overall, there is an increasing trend and a steep jump when gamma equal to 1.

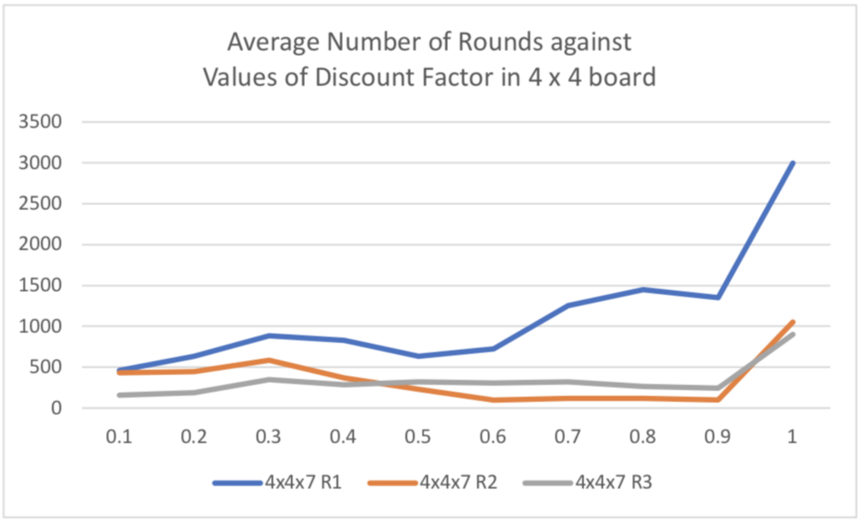


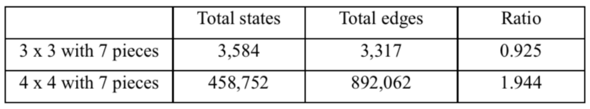
Figure 12: Graph of Average Number of Rounds against Values of Discount Factor in 4 x 4 board

On the other hand, similar to the 3 x 3 board, the trend of the number of rounds is similar to that of scores. The number of rounds also attain the highest with gamma value 1. And, there is also a crest found when gamma values 0.3. In the meantime, the curve of both graphs is similar to each other. Hence, it is also proved that the scores are directly proportional to the number of rounds survived.

From the above two groups, it showed that applying reward function 1 will get the highest score and survive the largest number of rounds under any values of gamma. Meanwhile, it is easy to discover that the difference between scores and the number of rounds with reward function 1 and other two reward function is quite large. Reward function 1 depends partially on the number of line killed and the number of possible moves for the state s'. It is evidential to gain the highest scores and have the longest survival time.
For games with seven pieces, the highest score would be obtained by applying reward function 1 with the value of gamma 1. Also, the relationship of the number of rounds and scores is discovered as directly proportional.

7 Challenge raised by MDP

In this project, the most difficult part is to overcome the memory issue. Since the configuration of a larger board, 5x5, is too larger that it fails to find out optimal policy by MDP.



	Total states	Total edges	Ratio
3 x 3 with 7 pieces	3,584	3,317	0.925
4 x 4 with 7 pieces	458,752	892,062	1.944

Figure 13: Number of states and edges in 3 x 3 and 4 x 4 boards

From the above data, we can see that with a larger board, the number of edges are increase with size. For 5 x 5 board, we try to estimates number of edge as 2 times of number of states. The number of state for 5 x 5 boards = $2^{25} * 7$

Assuming we need 4 bytes for storing each variables. For each state, we need to store:

- States (board with integer array size 5 x 5)
- Edges (number of action estimated with 3 times as the number of state, each has a tuple of int with size 2)
- Values (a float)
- Policies (a tuple of int with size 2)

we at least need $2^{25} * 7 * (5 * 5 + 3 * 2 + 1 + 2) * 4 = 2^{32.89}$ which is too large to be compute by using MDP

Therefore, the next step we need to do is apply reinforcement learning by allow the machine to learn in unknown environment instead of given all states for it to compute.

8 Comparison of different methodologies

To start our reinforcement learning in 1010!. We tried to study and compare different methodologies so as to find out the most suitable one for our game.

8.1 Why do we need Reinforcement Learning (RL)?

Reinforcement Learning is a machine learning technique that allow the agent to learn from an interactive environment. Previously, we solve Markov Decision Process (MDP) by dynamic programming. It is necessary to provide all set of states S, set of actions A, transition probabilities to next state $T(s, a, a')$, and the reward function R in order to calculate the Q-Values, hence finding out an optimal policy. However, it fails to calculate when the configuration of game is too large.(i.e. a huge number of states). Reinforcement learning is a model-free method to solve MDP without knowing transition model and reward. The agent can learn by exploring the environment [5]. There are two common model-free reinforcement learning algorithm: Monte-Carlo, Temporal-Difference.

8.2 Monte-Carlo (MC)

Monte-Carlo learns value function from episodes of experience where an episode is a sequance of steps, $(S_1A_1R_1), (S_2A_2R_2), (S_3A_3R_3)...(S_{t-1}A_{t-1}S_t)$ until the termination state. The value is updated by calculating the average reward of each step in a episode.[5]

$$v(s) = E[G_t|S_t = s] \qquad and \qquad G_t = R_{t+1} + \gamma R_{t+2}...$$

However, it is not applicable to our game as the game can last forever, it will not exist a terminal state for MC to compute.

8.3 Temporal-Difference (TD)

Comparing with Monte-Carlo, it can learn from incomplete sequences. This means that it is not necessary to wait until the end of the episode to update the value estimation. Instead, it can update at the current state by adding previous estimated Value $V(s)$ to the adjusted TD error [9].

$$V(s_t) \leftarrow V(s_t) + \alpha(R(s_{t+1}) + \gamma(s_{t+1}) - V(s_t))$$

where α is the learning weight, γ is the discount factor.

There are two common TD algorithm, State-Action-Reward-State-Action (SARSA) and Q-learning.

8.3.1 State-Action-Reward-State-Action (SARSA)

SARSA is to learns the Q-value according to the action performed by the current policy. This means that the next action a_{t+1} is decided at current state s_t and a_t. [3]

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

8.3.2 Q-learning

Q-learning is to estimate a state-action function for a target policy that select the action with highest reward. The policy is the action the contribute to the largest value. [3]

$$\text{Q(s,a)} \leftarrow Q(s,a) + \alpha(R + \gamma max'_a Q(s',a') - Q(s,a))$$
$$\pi \leftarrow \arg\max Q(s',a')$$

However, applying Q-learning, it needs to create a Q-table to store the state explored. Although size is highly reduced when comparing with MDP, eventually when it has enough exploration, the size of Q table is still very large that memory issue will also be happened.

8.4 Approximate Q-Value

In order to deal with this problem, we have to use a function approximation to solve it instead of store all states for computing. The most popular function approximation method is using neural network to help us estimate the Q-value.

8.4.1 Neural Network

Neural network is formed by a number of nodes and connected by links. There is a numeric weight associate between links. Each unit of node is connected in layers. Output of one nodes is the input of the next layer's nodes. Neural network learns by updating the weights between links. The network takes a state s as the input and return the estimated value of each action by transforming the weighted sum of nodes with activation function into final value. [8] The action corresponding to the highest value returned will be chosen as the policy π of state s.

8.4.2 Least Square Policy Iteration (LSPI)

Another common algorithm to approximate values is to use Least Square Policy Iteration. It is an algorithm aiming to estimate the value with a linear equation. The equation is the weighted sum of several features function[7].

$$Q^{\pi}(s,a) = w_0 + w_1\phi_1(s,a) + w_1\phi_1(s,a) + ... + w_n\phi_n(s,a) = \sum_{i=1}^{k} \phi_i(s,a)w_j \qquad (1)$$

Comparing with neural network, it is much simpler to solve the problem with a linear architecture by LSPI. Moreover, it is easier to monitor the weighting of the features as those functions are designed by us. At the same time we can point out which feature is more important to the game. With LSPI, we can implement the architecture by ourselves to find out the game strategy, instead of using such complex one in neural network. It is worth to use a simpler algorithm as a starting point of doing reinforcement learning on our game 1010! in this project. Therefore, we decided to apply LSPI to solve our problem. The details of what is LSPI and how to apply it on 1010! will be discussed below.

9 Least Square Policy Iteration in 1010!

9.1 Least-Square Approximation Algorithm

In this algorithm, the value is approximated from values of feature multiplied by weights.

$$Q^{\pi}(s,a) = \sum_{i=1}^{k} \phi_i(s,a) w_j \tag{2}$$

where $Q^{\pi}(s,a)$ is the estimated value for state s with action a, which is calculated by linear parametric combination of n feature function. $\phi_n(s,a)$ is the n^{th} feature values corresponding to n feature function, w_n is weight of the n^{th} feature function. In our case, we have $n = 21$ features.

Now let L be the sample size, k be the number of features, and

$$\mathbf{Q} = \mathbf{\Phi}^T \mathbf{w} \tag{3}$$

where $\mathbf{Q}$ is a matrix with size $L \times 1$ containing all Q-value of L samples, $\mathbf{\Phi}$ is a matrix with size $L \times k$ containing all ϕ vector of L samples, each ϕ vector contain k value of features of that sample, $\mathbf{w}$ is a matrix with size $k \times 1$ containing numeric weight values corresponding to each feature function. [4]

9.2 Least-Square Fixed-Point Approximation

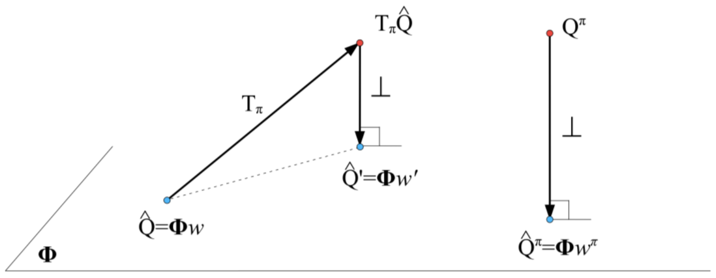


Figure 14: Policy evaluation and projection methods

In order to have a good approximation, we hope that $T_\pi \hat{Q}^\pi \approx Q^\pi$. By applying orthogonal projection [4] to minimize the L_2 norm for updating Q^π, we have

$$\begin{aligned}\hat{Q}^T &= \Phi(\Phi^T\Phi)^{-1}\Phi^T(T_\pi \hat{Q}^\pi)\\ \hat{Q}^T &= \Phi(\Phi^T\Phi)^{-1}\Phi^T(R + \gamma \mathbf{P}^\pi \Pi_\pi \Phi w^\pi)\end{aligned}$$

By substituting, $\hat{Q} = \Phi w^{\pi}$, we have

$$\begin{aligned}
\Phi(\Phi^T\Phi)^{-1}\Phi^T(R + \gamma \mathbf{P}^{\pi}\Pi_{\pi}\Phi w^{\pi}) &= \hat{Q}^T \\
\Phi(\Phi^T\Phi)^{-1}\Phi^T(R + \gamma \mathbf{P}^{\pi}\Pi_{\pi}\Phi w^{\pi}) &= \Phi w^{\pi} \\
(\Phi^T\Phi)^{-1}\Phi^T(R + \gamma \mathbf{P}^{\pi}\Pi_{\pi}\Phi w^{\pi}) &= w^{\pi} \\
\Phi^T(R + \gamma \mathbf{P}^{\pi}\Pi_{\pi}\Phi w^{\pi}) &= (\Phi^T\Phi)^{-1} w^{\pi} \\
\Phi^T(\Phi - \gamma \mathbf{P}^{\pi}\Pi_{\pi}\Phi) w^{\pi} &= \Phi^T R
\end{aligned} \tag{4}$$

Therefore, we can represent equation (4) to a simpler one.

$$\mathbf{A} w^{\pi} = b$$

where

$$\begin{aligned}
\mathbf{A} &= \Phi^T(\Phi - \gamma \mathbf{P}^{\pi}\Pi_{\pi}\Phi) \\
b &= \Phi^T R
\end{aligned}$$

A and b can be updated by incremental rule below

$$\begin{aligned}
\mathbf{A}^{t+1} &= \mathbf{A}^t + \phi(s_t, a_t)(\phi(s_t, a_t) - \gamma\phi(s'_t, \pi(s'_t)))^T \\
b^{t+1} &= b^t + \phi(s_t, a_t) r_t
\end{aligned}$$

Weights of feature w^{π} can be solved by $w^{\pi} = \mathbf{A}^{-1} b$

Algorithm 1: LSPI Training algorithm

```
1 Initialize $w \leftarrow 0$;
2 Initialize $A \leftarrow 0$;
3 Initialize $b \leftarrow 0$;
4 generate sources of sample $S$ in $D$
5 For each $S(s, a, r, s')$ ∈D:
6     A $\leftarrow \mathbf{A} + \phi(s, a)(\phi(s, a) - \gamma\phi(s', \pi(s')))^T$
7     $b$ ←b+$\phi(s, a) r$
8 $w^{\pi} \leftarrow \mathbf{A}^{-1} b$;
9 **return** $w^{\pi}$
```

When the weight w is ready, from (2), we can calculate the approximate value by dot product of $\phi(s, a)$ and w for each state and action

$$\hat{Q}^{\pi}(s, a) = \phi(s, a) w \tag{5}$$

10 Applying LSPI into 1010!

Unlike the previous work, we are now extending the game up to the original scale.

10.1 State s

Same as last semester, we defined a state s of 1010! as a game board together with one of the pieces. Below is one of the example.

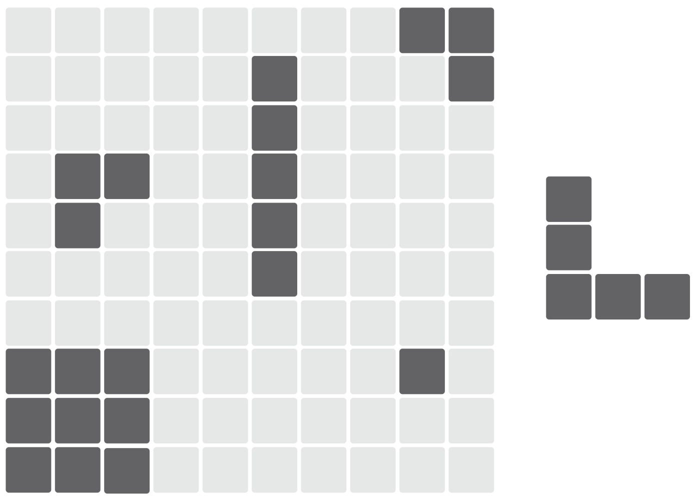
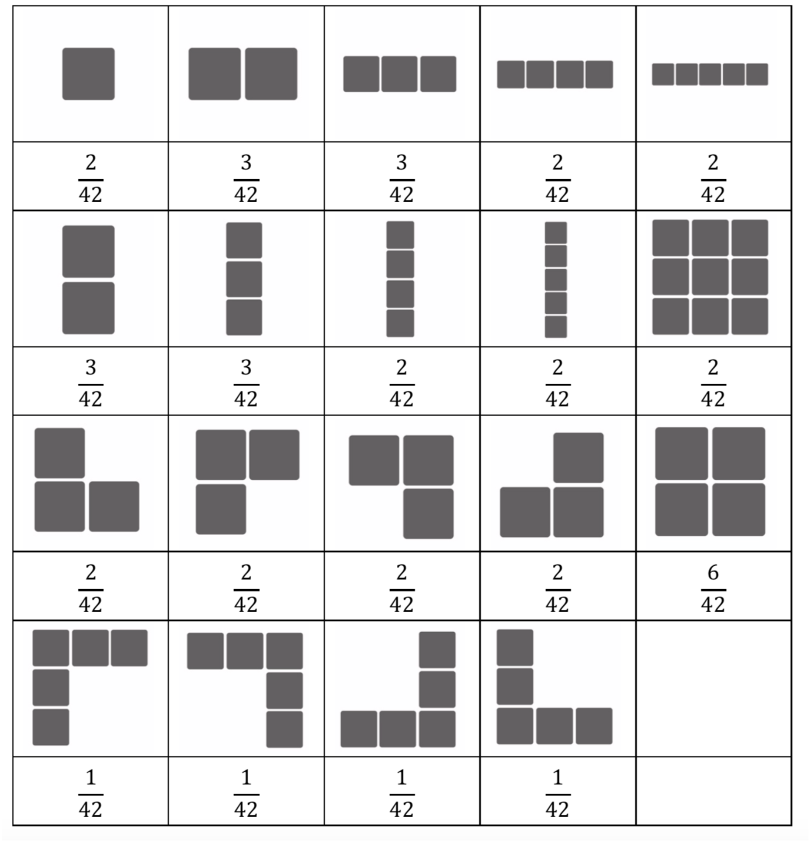


Figure 15: Example of a state: a game board with a random piece

10.2 Pieces π

Total we have 19 different pieces with probabilities as the original game. [1]



$\frac{2}{42}$	$\frac{3}{42}$	$\frac{3}{42}$	$\frac{2}{42}$	$\frac{2}{42}$
$\frac{3}{42}$	$\frac{3}{42}$	$\frac{2}{42}$	$\frac{2}{42}$	$\frac{2}{42}$
$\frac{2}{42}$	$\frac{2}{42}$	$\frac{2}{42}$	$\frac{2}{42}$	$\frac{6}{42}$
$\frac{1}{42}$	$\frac{1}{42}$	$\frac{1}{42}$	$\frac{1}{42}$	

Figure 16: shape of pieces with probabilities

10.3 Policy π

A contains all possibles move of a pieces,
π is the action selected from A which gives the highest value of $Q(s,a)$ among the others.

Policy π can be obtained by

$$\pi(s) = \arg\max \hat{Q}(s,a) = \arg\max \phi(s,a)^T w$$

10.4 Feature function

The input of the feature function is a game board s with the action s chosen. Total we have 21 feature functions. After calculating each values of each features, it then form a vector $\phi(s,a)$ with size $n = 21$.

$$\phi(s,a) = \begin{pmatrix} \phi_1(s,a) \\ \phi_2(s,a) \\ \vdots \\ \phi_n(s,a) \end{pmatrix}$$

ϕ	Feature Function Description
#1	average horizontal consecutive holes
#2	maximum horizontal consecutive holes
#3	average vertical consecutive holes
#4	maximum vertical consecutive holes
#5	number of occupied cell
#6	number of blank cell
#7	number of possible action of putting horizontal 5x1 pieces
#8	number of possible action of putting horizontal 1x5 pieces
#9	number of possible action of putting horizontal 3x3 pieces
#10	maximum area of square
#11	maximum area of rectangle
#12	distant from edge
#13	number of full blank horizontal line
#14	number of full blank vertical line
#15	number of component of blank space
#16	number of component of occupied space
#17	max blank space
#18	max full space
#19	reward when taking action a
#20	expected value of survival a
#21	bias with value 1 a

Table 1: Description of 21 feature functions

10.5 Integrating LSPI with 1010!

The game is trained in an unknown environment by exploration of states. By algorithm 1, the sample D is generated by playing game directly. All steps of each rounds are used to update $\mathbf{A}$ and b. For every 20 turns, we updated the weight function w. During the game, each time we select policy π with the highest $Q(s,a)$ calculated by (5). Repeat the game until weight function w converge. The penalty of game over state is set to be -1000.

Algorithm 2: Learning by LSPI algorithm in game 1010!

```
1  Initialize $w \leftarrow 0$;
2  Initialize $A \leftarrow 0$;
3  Initialize $b \leftarrow 0$;
4  play game and generate $S(s, a, r, s')$
5  For each state $s$ explored by the game:
6      Find all Action $A$;
7      For each $a$ in $A$:
8          calculate Q(s,a) by (5);
9          calculate reward $r$ of taking action $a$;
10     $\pi_a = \arg\max Q(s, a)$;
11     if $s'$ is an gameover state:
12         generate sample $S(s, \pi_a, -1000$,s');
13     else:
14         generate sample $S(s, \pi_a$,r,s');
15     update A and $b$ by algorithm 1
16     update $w$ for every 20 turns
17 **return** $w^\pi$
```

11 Result and Analysis

11.1 Comparison between MDP and LSPI

In the last semester, we have found out an optimal solution of a lower scale game with board size 4×4 and 7 different pieces with probabilities as below:

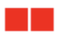
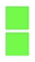
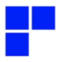
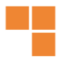
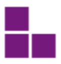
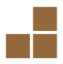
Pieces							
Probability	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{10}$	$\frac{1}{10}$	$\frac{1}{10}$	$\frac{1}{10}$

Figure 17: Example of a state: a game board with a random piece

	MDP	LSPI
r_1	1450.7	307.2
r_2	115.5	276.05
r_3	307.2	302.2

Table 2: number of average round between MDP and LSPI 4x4 board with 7 pieces in 10 turns

Recall that the reward function we decide in MDP

Reward Function r_1 : (number Of Line Killed)^2 + no. of possible moves of state s'
Reward Function r_2 : (number Of Line Killed)^2
Reward Function r_3 : no. of possible moves of state s'

From the result, we can see that with gamma equal to 0.8, the performance of LSPI is worse than MDP with reward function r_1. However, reward function r_2 LSPI have better performance then MDP and nearly same performance of r_3.

The key of MDP to provide an optimal policy is to define a suitable reward function. Therefore, we can see that among different reward functions, its performance varies in MDP. However, the key of LSPI to provide a good approximation is to design some suitable feature functions. Since the feature functions of the above three experiments remain unchanged, the performance is nearly the same no matter which reward function we used. In fact, according to the feature function we design in section 4.4, only ϕ_{19} is related to the reward we get, hence, in LSPI, the effect of reward function is not significant. Comparing with MDP, the performance of LSPI is worse than MDP when a worth reward function is designed. However, LSPI can still catch up the performance of MDP with when the reward function is not much well designed.
MDP can give better performance on the game, however, it fails to give optimal solution in a larger game. Therefore, it is not practical enough to find out the original game strategy. On the other hand, although LSPI cannot give optimal solution, it can still catch up performance of MDP and help to find out strategy of a larger scale of game.

11.2 Result of 10×10 board

11.2.1 Performance of LSPI

The result of training 4x4 board above shows that even though we apply different reward function, it nearly has no difference on performance. Therefore, the reward function we use to train 10×10 game is simply r_2, which is the number of lines killed.

By experiments, among 20 trials of the game, 15 trials is lose by randomising a pieces of 3×3 shape. In order to provide a more significant training of LSPI, we tried to remove this pieces from game. The performance of these two versions of game are given below.

	with 3×3 pieces	without 3×3 pieces
max	517	1047
avg	204.473	517.736

Table 3: Average rounds of 20 trials of game with and without 3×3 pieces

With the same feature functions in LSPI, these two versions of game show a huge different in performance. The maximum round of the game without 3×3 can even reach more than thousand rounds. This shows that 3×3 pieces is relatively large comparing with the board which much increase the difficulty of the game. As a result, LSPI is not powerful enough to find a good policy for placing this piece by the feature functions we designed. However, the performance of game without 3×3 piece still shows the training effect of LSPI.

11.2.2 Interpretation of weights of feature

A higher weight value of an feature means that it is worth to have a higher value of that feature. Features with weighting nearly 0 is said to be relatively not much important for the game.

All together we designed 21 different feature functions. The weights is as follow:

ϕ	Feature Function Description	weight value
#1	average horizontal consecutive holes	45.232
#2	maximum horizontal consecutive holes	-0.37348
#3	average vertical consecutive holes	39.333
#4	maximum vertical consecutive holes	1.7828
#5	number of occupied cell	2.1548
#6	number of blank cell	4.6361
#7	number of possible action of putting horizontal 5x1 pieces	-3.6278
#8	number of possible action of putting horizontal 1x5 pieces	-3.1456
#9	number of possible action of putting horizontal 3x3 pieces	-1.88584
#10	maximum area of square	0.43437
#11	maximum area of rectangle	0.46786
#12	distant from edge	-0.161435
#13	number of full blank horizontal line	1.3728
#14	number of full blank vertical line	-0.37325
#15	number of component of blank space	-6.5215
#16	number of component of occupied space	0.12630
#17	max blank space	-0.45505
#18	max full space	-0.16362
#19	reward of action a (prop. to number of line killed) a	0.18772
#20	expected value of survival a	1285.25
#21	bias with value 1 a	-1669.6

Table 4: Feature function with corresponding weight value

From the above result, the highest weight value is contributed by the expected value of survival with action a, ϕ_{20}, which is 1285.25. That is reasonable as the penalty of game over is set to be -1000 with it comes to a game over state. Once the game terminated, it will update the weight with a negative r. Hence, it will adjust the weight of the weight vector w so as to minimize the chances of the same case to occur again. As a result, the weighting of expected value of survival is trained to be large so that it can prevent game over.

Besides, the second highest values is contributed by ϕ_1 and ϕ_3, the average horizontal and vertical consecutive holes. Since the random pieces are in different shapes, it is better for it to reserve spaces for the future rounds. As a result, it ϕ_1 and ϕ_3 both have a high weighting.

Another a relatively larger weight is contributed by ϕ_{15}, number of component of blank space, which is a negative value of -6.5215. Number of component is simply the distribution of the blank space, that means the blank space are separated by the occupied cells which lowering the reserved area, hence the chances for pieces to put on. In order to minimize the distribution of the blank space, there is a negative weighting on this feature.

However, from our discovery by MDP in the last semester, it points out that maximizing the number of line killed and number of possible moves of next state is the most important feature. Comparing with LSPI, for ϕ_{19}, reward when taking action a, it weighting is just 0.18772, which is nearly zero. Beside, ϕ_7, ϕ_8, ϕ_8, which are related to the number of action of pieces, is even in a negative value. This show that LSPI found that killing line and computing just the number of actions is not such important to win the game.

12 Conclusion

In first part of the project, we applied Markov Decision Process to the game 1010!. Since solving MDP by dynamic programming required to calculated from all states, when the configuration of the game is too large, it fails to compute due to lack memory space. Therefore, we could only find out optimal policy in a lower scale of game.

In order to deal with this problem, we applied reinforcement learning, Least-square Policy Iteration on the game to approximate the value by a linear function.

Comparing to the performance between MDP and LSPI, in terms of performance, MDP can provide us an optimal policy with high performance when a reward function is well defined, if not, the performance of LSPI can also catch it up. Besides, MDP is depending on the quality of reward function while LSPI is depending on the quality of feature functions.

In LSPI, we designed 21 feature functions to approximate the value. From the result, we found that the most important feature is the expected value of survival chances of next state. Also, there is a trend that features related to reserving large area of spaces are also important to the game such as the average consecutive holes and number of component of blank space.

Unlike the result from last semester finding that the number of lines killed and the number of action can be taken for next state are the most important key to play the game by MDP. However, from the results of feature function in LSPI, the survival probability is the most important while line killing and number of actions are not significantly contributed to the value.

Although LSPI can already help us to enlarge the scale of game up to the original version of game and give us a reasonable performance, the result of implementing the full game shows that it is hard to use a linear function to approximate value and gives the best action for the game. The complexity of game board is too hard for human to think of a powerful linear function to solve it.

If we wish to get a better performance of this game, linear function is not enough to solve this game. It is believed that the performance of the game can be enhanced by other reinforcement learning method with non-linear/more complex architecture such as neural network. However, only using a simple linear function, LSPI is already powerful for training game up to a certain hardness, like what we did for 10×10 board without one of the pieces.

13 Division Of labour

13.1 Term 1

1. Research on existing game with different methods : Kong Sai Kwan
2. Research on How MDP is worked : Tang Sin Yee, Kong Sai Kwan
3. Implementing three greedy algorithm: Tang Sin Yee
4. Implementing and integrate the game 1010! with MDP : Tang Sin Yee
5. Visualization of game 1010! on Xcode with scale 3 x 3 and 4 x 4: Tang Sin Yee
6. Result Analysis : Kong Sai Kwan

13.2 Term 2

1. Study of different methodologies of reinforcement learning
- Monte-Carlo (MC) : Kong Sai Kwan
- Temporal-Difference (TD): Kong Sai Kwan
- Q-learning and Neural Network: Tang Sin Yee
- Least Square Policy Iteration (LSPI): Tang Sin Yee

2. Code division:
- Implement and integrate the game 1010! with LSPI algorithm: Tang Sin Yee
- Design and implement the feature functions for LSPI alogithm: Kong Sai Kwan
- Visualization of game 1010! on Xcode with scale 10 x 10: Tang Sin Yee

3. Train and adjust the feature function: Tang Sin Yee, Kong Sai Kwan
4. Result Analysis : Tang Sin Yee, Kong Sai Kwan

References

[1] Fabien Coelho. *1010! Analysis.* 2016. URL: http://blog.coelho.net/games/2016/07/28/1010-game.html.

[2] Adrien Lucas Ecoffet. *Beat Atari with Deep Reinforcement Learning! (Part 1: DQN).* 2017. URL: https://becominghuman.ai/lets-build-an-atari-ai-part-1-dqn-df57e8ff3b26e.

[3] Steeve Huang. *Introduction to Various Reinforcement Learning Algorithms. Part I (Q-Learning, SARSA, DQN, DDPG).* 2018. URL: https://towardsdatascience.com/introduction-to-various-reinforcement-learning-algorithms-i-q-learning-sarsa-dqn-ddpg-72a5e0cb6287.

[4] Ronald Parr. Michail G. Lagoudakis. *Least-Squares Policy Iteration.* 2003. URL: http://www.jmlr.org/papers/volume4/lagoudakis03a/lagoudakis03a.pdf.

[5] Madhu Sanjeevi. *Ch 12.1:Model Free Reinforcement learning algorithms (Monte Carlo, SARSA, Q-learning).* 2018. URL: https://medium.com/deep-math-machine-learning-ai/ch-12-1-model-free-reinforcement-learning-algorithms-monte-carlo-sarsa-q-learning-65267cb8d1b4.

[6] Madhu Sanjeevi. *Ch:13: Deep Reinforcement learning—Deep Q-learning and Policy Gradients (towards AGI).* URL: https://medium.com/deep-math-machine-learning-ai/ch-13-deep-reinforcement-learning-deep-q-learning-and-policy-gradients-towards-agi-a2a0b611617e.

[7] Daniel Seita. *Going Deeper Into Reinforcement Learning: Understanding Q-Learning and Linear Function Approximation.* 2016. URL: https://danieltakeshi.github.io/2016/10/31/going-deeper-into-reinforcement-learning-understanding-q-learning-and-linear-function-approximation/.

[8] Peter Norvig Stuart Russell. *Artificial Intelligence, A Modern Approach, Third Edition.* Pearson, 2010.

[9] Andre Violante. *Simple Reinforcement Learning: Temporal Difference Learning.* URL: https://medium.com/@violante.andre/simple-reinforcement-learning-temporal-difference-learning-e883ea0d65b0.