Assignment as Term Paper in Artificial Intelligence, BITS Pilani, India

Tic-Tac-Toe using Reinforcement Learning (Q-learning)

Atharva Mehta 2018ABPS0485P

Pushpan 2018A2PS0902P

Birla Institute of Technology and Science,
Pilani, 333031, Jhunjhunu, Rajasthan, India

Abstract

This report applies the reinforcement learning algorithm Q-learning to the simple pen and pencil game cross & zero, also known as Tic-Tac-Toe. The goal is to analyze how different parameters and opponents affect the performance of the Q-learning algorithm. Simple computer opponents play against the Random Q-learning algorithm and later Q-learning algorithms higher level experience. Several different parameters are tried and the results are plotted on detailed graphs. The graphs show that the best results are gained against other Q-learning algorithms and a high discount factor. When two Q-learning algorithms with different experiences played against each other, the best results were obtained with a learning rate of 0.5. The best results were shown when letting a Q-learning algorithm play against another Q-learning algorithm. A learning rate and discount factor of 0.5 and 0.96 showed the best results, however the data suggested even higher values of the discount factor might give even better results. In a Tic-Tac-Toe, it requires about 1 million games to reach a good result.

1. Introduction

This report is part of the CS F207 Artificial Intelligence course, project in Computer Science, first level, held by BITS Pilani. The goal of this project is to create a computer player using the Q-learning algorithm that will learn to play the traditional board game Tic-Tac-Toe. The computer player will then be evaluated in terms of efficiency and performance against multiple opponents with different strategies.

Artificial intelligence and machine learning are the growing field with increasing importance. It is used when normal algorithms are too difficult and complex to develop and because it has the ability to recognize complex patterns that might otherwise be overlooked. Machine learning methods can also be used when the activity to be performed is in a changing environment where adaptability is important to overcome new obstacles that may arise. An example of an area where machine learning has proven very effective is speech recognition. Human beings find it easy to recognize and interpret speech, but it is difficult to describe exactly how it is done. Just as a human being learns to understand a language, a computer can learn the same thing, through the use of machine learning methods. Other examples of areas where machine learning is important are stock market analysis, medical diagnostics, and facial recognition.

Purpose

Tic-Tac-Toe is a traditional board game that is normally played by two people at a time. Oftentimes, you don't have anyone to play with, so you need an opponent to play the role of a computer gamer. To make the game fun, the computer gamer must be properly skilled. By using a simple reinforcement learning algorithm, called Q-learning, to create a computer player, the goal is to analyze the performance and effectiveness of that player against different opponents. The report examines how these adversaries affect and improve the progression speed and end result of the Q-learning computer player.

Terminology

- **Tic-Tac-Toe**- A traditional board game usually played with paper and pencil
- **Reinforcement learning-** Reinforcement learning is a subset of the machine learning techniques. These techniques learn through trial and error in search of an optimal solution.
- **Q-learning**- Q-learning is a simple reinforcement learning algorithm
- **Agent-** An agent is a player in the game implementation, either a computer or an interface towards humans.
- **State** -The state of the environment; the current game configuration.
- **Action-** An action is a transition from one state to another, which is what a player does or can do in a specific state.
- **Feedback**- An action generates feedback from the environment. The feedback can be either positive or negative. The feedback is used in the Q-learning algorithm for estimating how good an action is. The term reward is used for positive feedback and negative feedback is called punishment.
- **Learning rate**- The learning rate is a parameter in the Q-learning algorithm, describing the relationship between old and new memories. A high value means that new memories take precedence over old memories and vice versa. A value of zero means that nothing will be learnt.
- **Discount factor**- The discount factor is a parameter in the Q-learning algorithm, affecting the importance of future feedback. A high discount factor increases the importance of future feedback.

Theory

The first player, who will be designated "X", has 3 possible strategically distinct positions to score during the first round. Superficially, it may appear that there are 9 possible positions, corresponding to the 9 squares of the grid. However, as we turn the board, we will find that in the first round each corner mark is strategically equivalent to every other corner mark. The same goes for any mark on the edge (middle side). From a strategic point of view, there are therefore only three possible first signs: corner, edge or center. Player X can win or force a tie from any of these starting marks; however, playing the corner gives the opponent the smallest choice of squares that must be played to avoid losing. This may suggest that the corner is the best opening shot for X, but another study shows that if the players are not perfect, a central opening shot is best for X. The second player, who will be designated. " O ", must reach the opening mark of X in order to avoid the forced victory. Player O must always respond to a corner opening with a center mark and a center opening with a corner mark. An open edge must be answered with a center mark, a corner mark next to the X or a mark on the edge opposite the X. Any other answer will allow X to force victory. Once the opening is over, O's job is to follow the list of priorities above to force the draw, or to get a win if X makes a weak play.

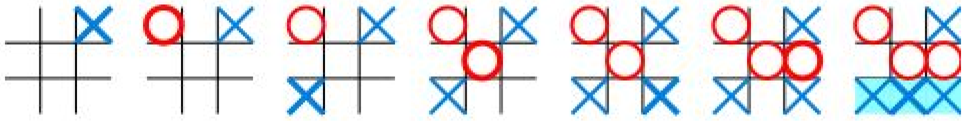


Figure 1. A scenario of game

So, the number of states obtained are = 3^9 as there are 9 spots and for each spot there are 3 outcomes possible X, O, - so we have 20K states possible and mean no. of possible moves is 4 so average state action pair is 20K x 4 = 80K state action pair which is not much considering the scope of Q-learning.

Q-learning is a reinforcement learning algorithm. The main advantage of this algorithm is that it is simple and therefore easy to implement; the current algorithm consists of a single line of code. To understand Q-learning, it is essential to understand how a problem is represented. It is possible to divide almost any problem into several situations, the so-called states. For example, in the Tic-Tac-Toe game, there is a state for every possible grid configuration, i.e. every different set of crosses and zeros. In each state a few actions can be performed, one action corresponds to which moves are legal in the game. The simplest form of Q-learning stores a value for each state-action pair in a matrix or table. The fact that it requires a value for each state-action pair is one of the drawbacks of learning Q; it requires a huge amount of memory. One possible solution to this problem is to use an artificial neural network, although this approach is not covered in this report. In each state visited by the algorithm, it checks the best possible action it can take. To do this, it first checks the Q value of each state it has a chance to access in a single step. It then takes the maximum of these future values and incorporates them into the current Q value. When a return is provided, the only value that is updated is the Q value corresponding to the pair of state actions that provided the return in question. The algorithm for updating Q values is shown in the following equation.

$$Q(s_t,a_t) \leftarrow \underbrace{Q(s_t,a_t)}_{\text{old Q-value}} + \underbrace{\alpha}_{\text{learning rate}} \times \Big[\overbrace{\underbrace{r_{t+1}}_{\text{feedback}} + \underbrace{\gamma}_{\text{discount factor}} \underbrace{\max_a Q(s_{t+1},a)}_{\text{max future Q-value}}}^{\text{expected discounted feedback}} - \overbrace{Q(s_t,a_t)}^{\text{old Q-value}} \Big]$$

The fact that only one value is updated when feedback is provided gives Q-learning an interesting property. It takes some time for the feedback to propagate back through the matrix. The next time a certain state-action pair is evaluated, the value will be updated for the states it can lead to directly. This means that in order for the feedback to propagate back to the beginning, the same path must be taken the same number of times that the paths are long, with each new iteration on the path propagating the effects one step back. 8 Q-learning for a simple board game If the algorithm has always taken the best path it knows, it is very likely that it will end up with a local maximum where it will always follow the same path. Even if the path he takes is the best he knows, that doesn't mean there isn't a better path. To counter this it is necessary to let the algorithm sometimes take a new path and not the best one, so that it will hopefully find a better solution. If there is one and only one stable solution, i.e. which action is best in each state, the Q-learning algorithm converges to this solution. Due to the backward propagation property of Q-learning, however, this requires a large number of visits to each possible pair of state actions. The algorithm has two parameters, the learning rate α and the discount factor γ. These two factors affect the behavior of the algorithm in different ways. The learning rate decides the number of future actions to be considered. A learning rate of zero would mean that the agent would not learn anything new and a learning rate of one would mean that only the most recent information is taken into account. The discount factor determines how much future feedback is taken into account by the algorithm. If the discount factor is zero, what happens in the future is ignored. Conversely, when the discount factor approaches one, priority will be given to a long-term high reward.

2. Implementation

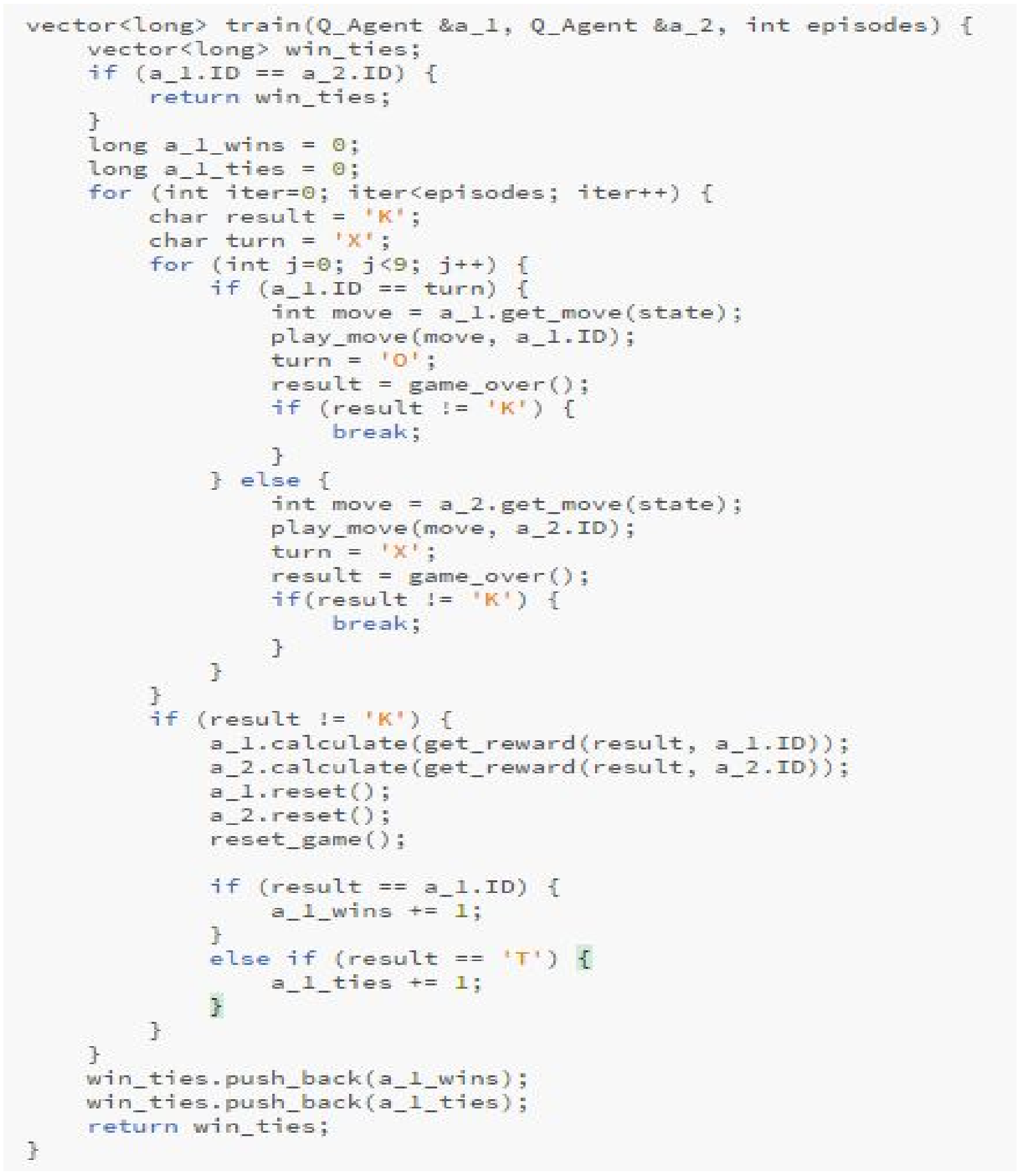
This Assignment is carried out using C++ language. To plot the graphs, Python Matplotlib library is used as doing it C++ is infeasible. The Code includes classes for each of the agents i.e. Q-Agent, Random Agent and Simple Agent and Final class is for the Game implementation.

- **Game Implementation and Representation**

  As the game consists of 9 locations to put X or O, so rather than forming a multi-dimensional matrix we have used string to represent the 9 possible locations to put X or O. The TicTacToe class consists of states as variables which store the present state. Class contains

  - Reset function to reset the state again for next episode.

- Game-over function to check whether the game has come to an end or not. If the game has ended, it returns a character based on whether the game is won by someone or has tied.
- Play-move function so that the Agent can make a move.
- Train function is there to train our agent against the other agent. This function facilitates the game play between the agents that are required to play. It takes an object of the agent which is playing and batch size as the input and It returns the number of wins and ties based on batch size fed to it. Figure 2 shows the snippet of the code. Code also calls various methods of the agent's class.

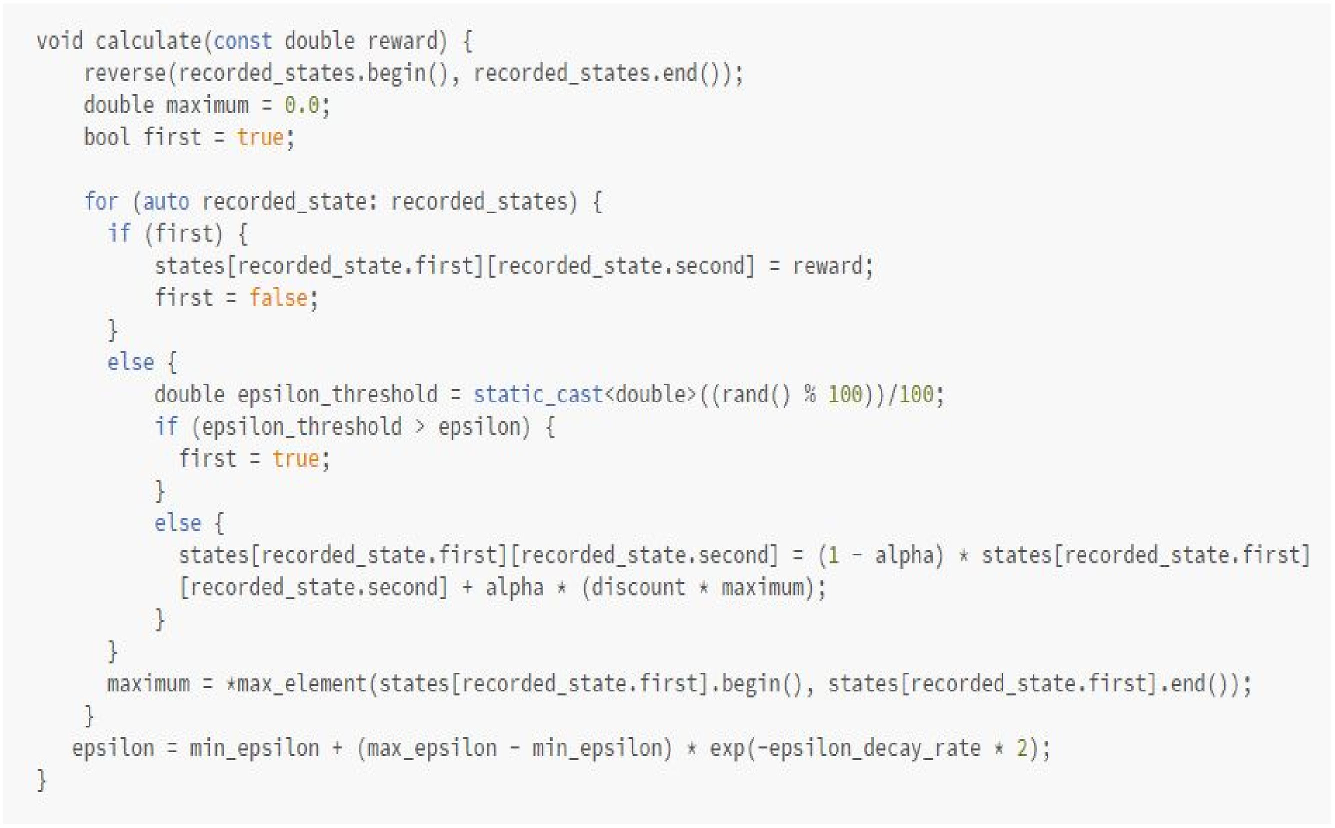


```
vector<long> train(Q_Agent &a_1, Q_Agent &a_2, int episodes) {
    vector<long> win_ties;
    if (a_1.ID == a_2.ID) {
        return win_ties;
    }
    long a_1_wins = 0;
    long a_1_ties = 0;
    for (int iter=0; iter<episodes; iter++) {
        char result = 'K';
        char turn = 'X';
        for (int j=0; j<9; j++) {
            if (a_1.ID == turn) {
                int move = a_1.get_move(state);
                play_move(move, a_1.ID);
                turn = 'O';
                result = game_over();
                if (result != 'K') {
                    break;
                }
            } else {
                int move = a_2.get_move(state);
                play_move(move, a_2.ID);
                turn = 'X';
                result = game_over();
                if(result != 'K') {
                    break;
                }
            }
        }
        if (result != 'K') {
            a_1.calculate(get_reward(result, a_1.ID));
            a_2.calculate(get_reward(result, a_2.ID));
            a_1.reset();
            a_2.reset();
            reset_game();

            if (result == a_1.ID) {
                a_1_wins += 1;
            }
            else if (result == 'T') {
                a_1_ties += 1;
            }
        }
    }
    win_ties.push_back(a_1_wins);
    win_ties.push_back(a_1_ties);
    return win_ties;
}
```

Figure 2. Train Function in TicTacToe Class

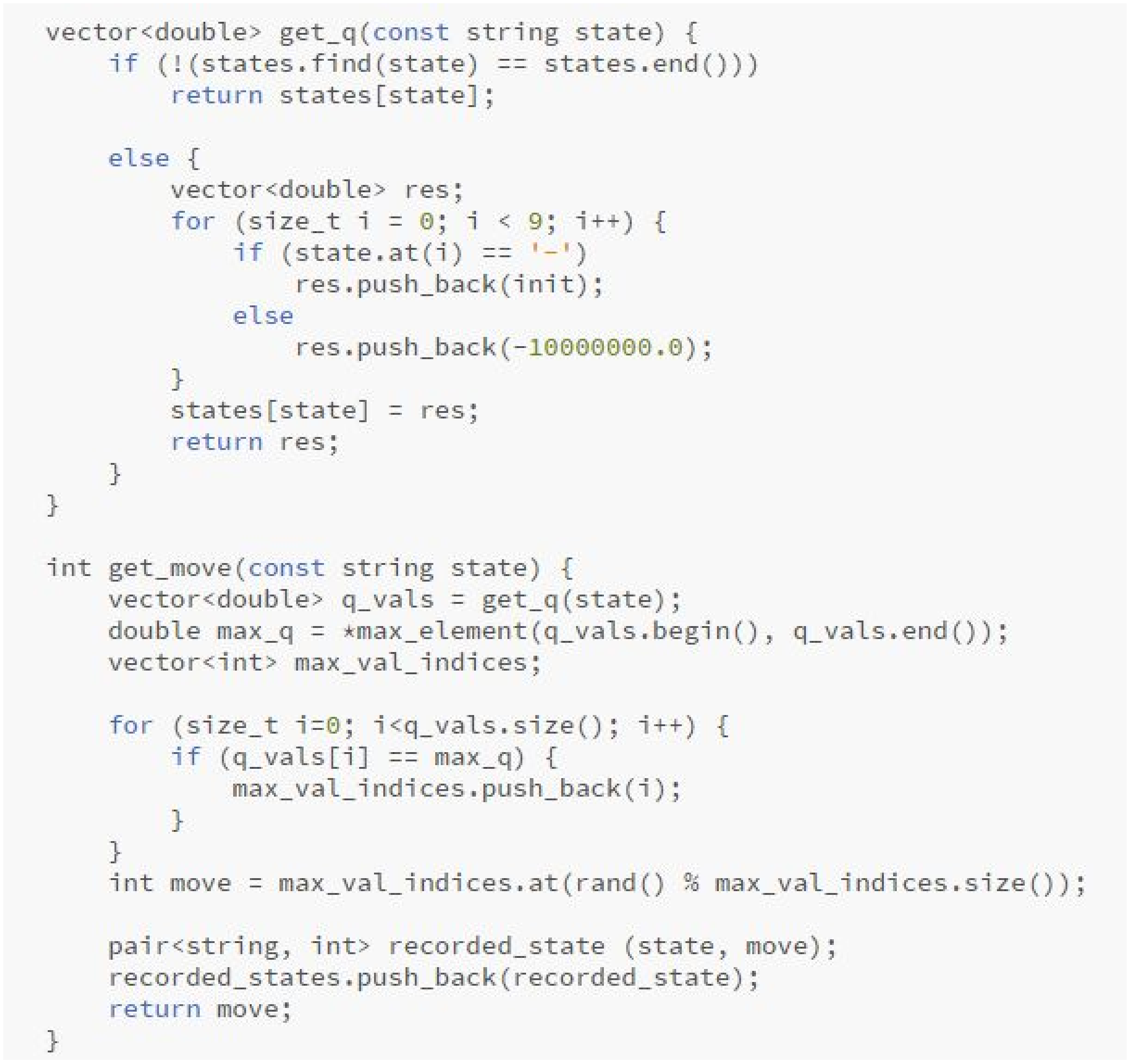
- **Q-Agent Class**

  In the Q-Agent class, the agent begins by first exploring the environment as it has no idea about the state of the environment and gradually begins exploitation. This is taken into account by ensuring the exploration rate variable ε (which controls the trade-off between exploration and exploitation) is set to 1 in the first episode. As the training progresses and in each subsequent episode, we apply exponential decay to ε to favour more exploitation and less exploration in the subsequent episodes. This exponential decay and the Q-value update is implemented in the “calculate” function. The function “get_q” returns the Q-values for a particular state and the function “get_move” returns the final move (a mix of exploration and exploitation depending on ε) to the “train” function of the TicTacToe (environment) class.



```
void calculate(const double reward) {
    reverse(recorded_states.begin(), recorded_states.end());
    double maximum = 0.0;
    bool first = true;

    for (auto recorded_state: recorded_states) {
      if (first) {
          states[recorded_state.first][recorded_state.second] = reward;
          first = false;
      }
      else {
          double epsilon_threshold = static_cast<double>((rand() % 100))/100;
          if (epsilon_threshold > epsilon) {
            first = true;
          }
          else {
            states[recorded_state.first][recorded_state.second] = (1 - alpha) * states[recorded_state.first]
            [recorded_state.second] + alpha * (discount * maximum);
          }
      }
      maximum = *max_element(states[recorded_state.first].begin(), states[recorded_state.first].end());
    }
   epsilon = min_epsilon + (max_epsilon - min_epsilon) * exp(-epsilon_decay_rate * 2);
}
```

Figure 3. Calculate Function in Q-Agent Class

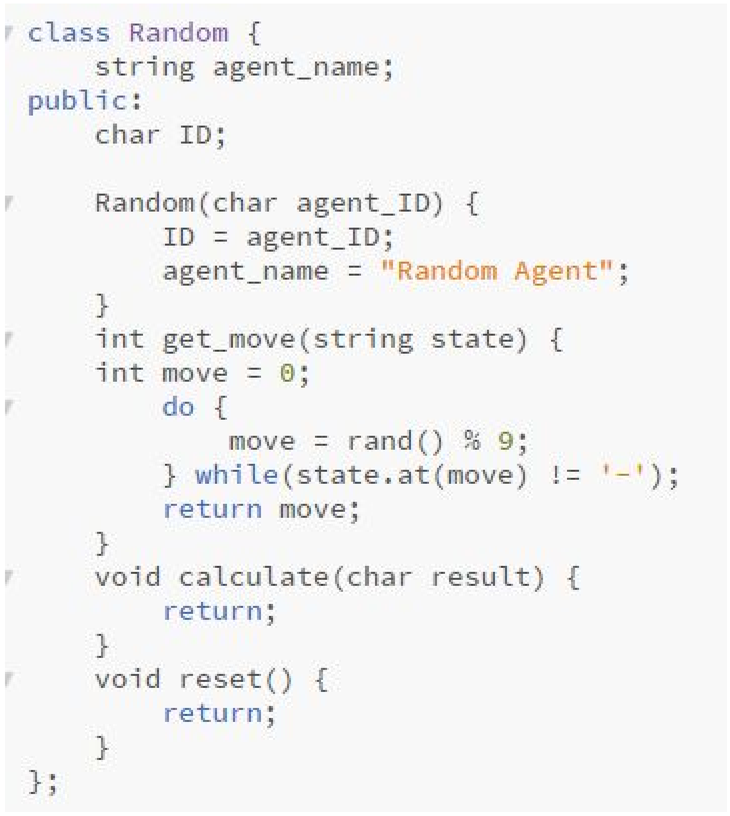


```
vector<double> get_q(const string state) {
    if (!(states.find(state) == states.end()))
        return states[state];

    else {
        vector<double> res;
        for (size_t i = 0; i < 9; i++) {
            if (state.at(i) == '-')
                res.push_back(init);
            else
                res.push_back(-10000000.0);
        }
        states[state] = res;
        return res;
    }
}

int get_move(const string state) {
    vector<double> q_vals = get_q(state);
    double max_q = *max_element(q_vals.begin(), q_vals.end());
    vector<int> max_val_indices;

    for (size_t i=0; i<q_vals.size(); i++) {
        if (q_vals[i] == max_q) {
            max_val_indices.push_back(i);
        }
    }
    int move = max_val_indices.at(rand() % max_val_indices.size());

    pair<string, int> recorded_state (state, move);
    recorded_states.push_back(recorded_state);
    return move;
}
```

Figure 4. Get-Q and Get-move Function in Q-Agent Class

- ## Random Agent Class

Random Agent Class rather than doing both exploration and exploitation as Q-Agent was doing it is focusing completely on exploration i.e. its moves are completely random. It contains only one function ``get _move”. This function takes the current state as the input and returns the random move that the agent performs. Figure 5 shows the snippet of the code for the class.

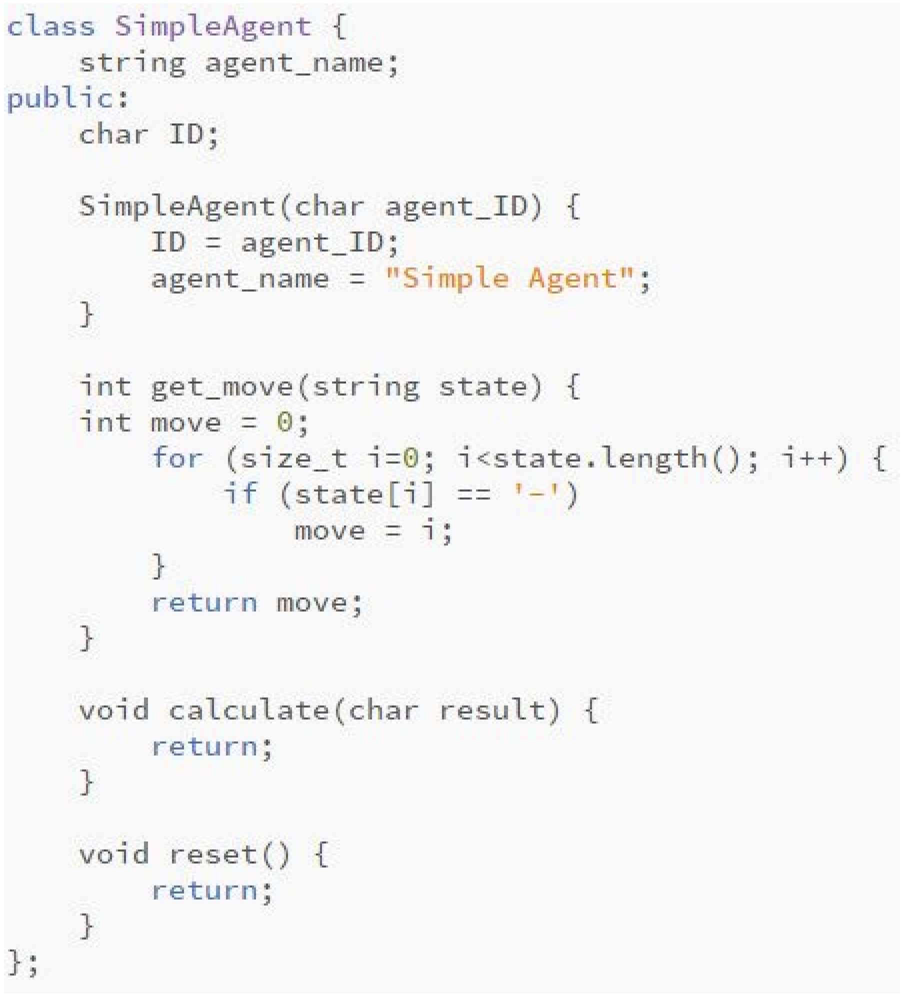


```
class Random {
    string agent_name;
public:
    char ID;

    Random(char agent_ID) {
        ID = agent_ID;
        agent_name = "Random Agent";
    }
    int get_move(string state) {
    int move = 0;
        do {
            move = rand() % 9;
        } while(state.at(move) != '-');
        return move;
    }
    void calculate(char result) {
        return;
    }
    void reset() {
        return;
    }
};
```

Figure 5. Random Class

- ## Simple Agent Class

Similar to what a random class is doing, a Simple Agent is taking one particular action every time without focusing on the outcome it is going to give. It always chooses the maximum unoccupied labeled position. This class contains only one working function which takes the same as Random Agent class' "get_move" function, current state and returns the location the move to make. Figure 6 shows the snippet of the class code.



```
class SimpleAgent {
    string agent_name;
public:
    char ID;

    SimpleAgent(char agent_ID) {
        ID = agent_ID;
        agent_name = "Simple Agent";
    }

    int get_move(string state) {
    int move = 0;
        for (size_t i=0; i<state.length(); i++) {
            if (state[i] == '-')
                move = i;
        }
        return move;
    }

    void calculate(char result) {
        return;
    }

    void reset() {
        return;
    }
};
```

Figure 6. Simple Agent Class

3. Result

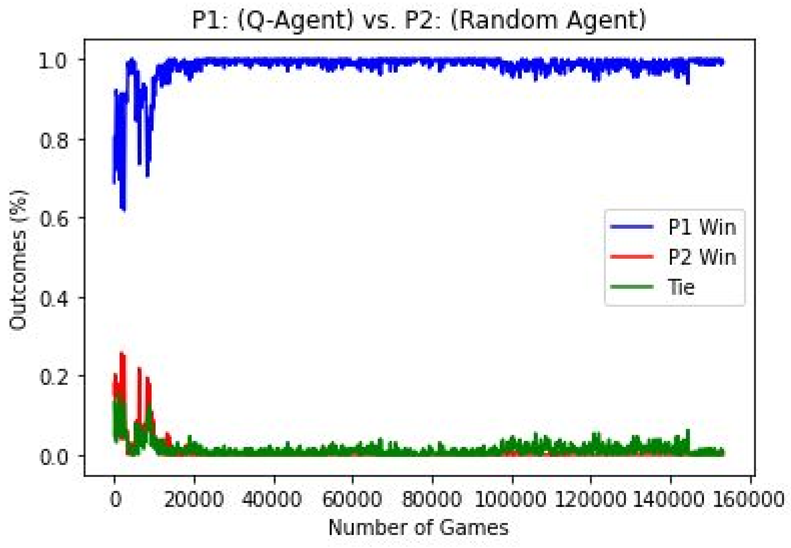
Our analysis is only on the few following values of learning rate (α), discount factor (γ), and exploration rate (ε).

In the exponential decay of the exploration rate (ε), the maximum exploration rate was taken as 1 and minimum exploration rate was set to 0.01.

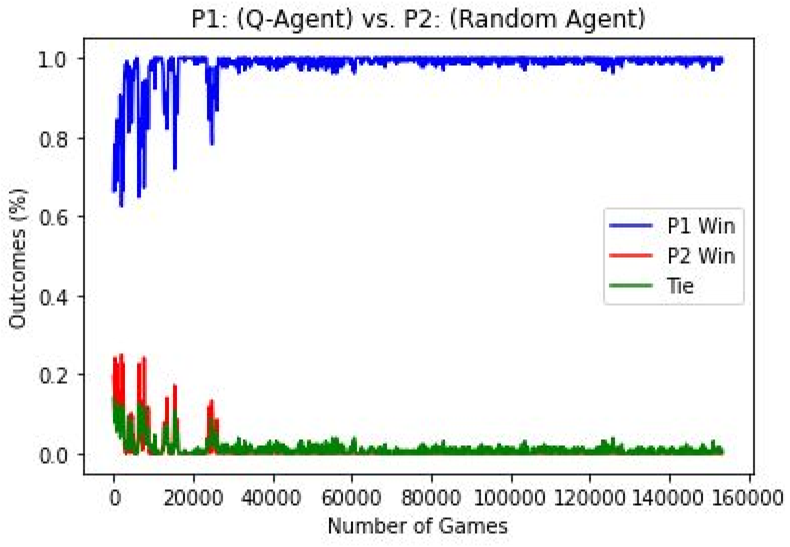
Learning Rate (α)	0.9	0.7	0.5
Discount Factor (γ)	0.96	0.85	0.71

Exploration Rate Decay	0.01	0.05	0.15

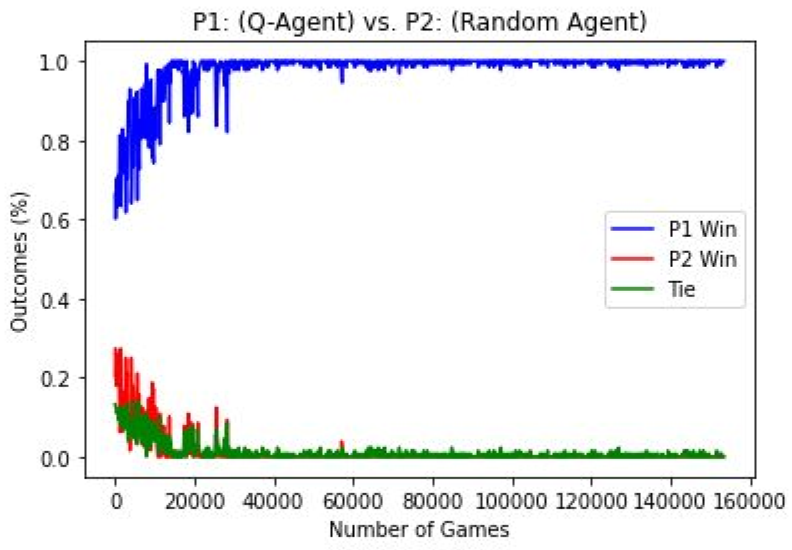
- **Q1: Q-Agent vs Random Agent**



α = 0.5, ᵞ = 0.96, exploration decay rate = 0.01



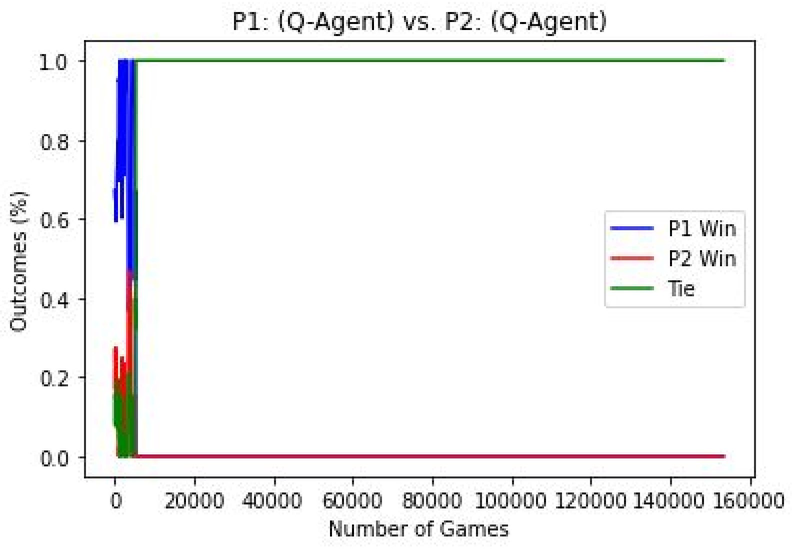
α = 0.7, ᵞ = 0.85, exploration decay rate = 0.05



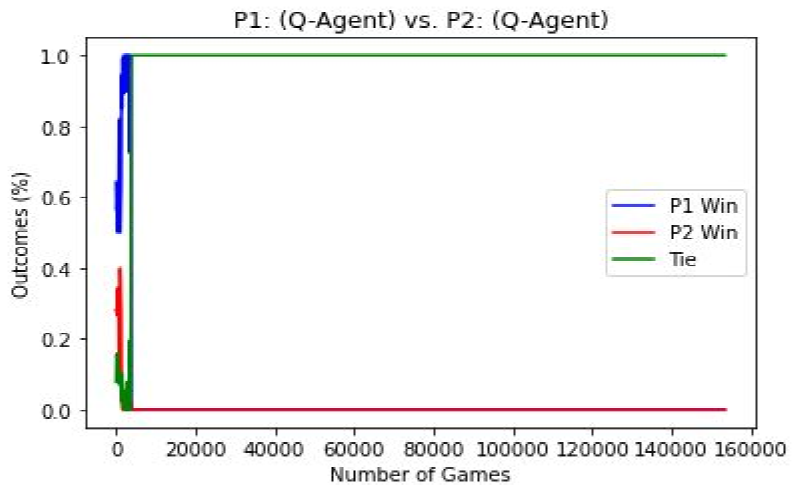
α = 0.9, ᵞ = 0.71, exploration decay rate = 0.15

The low value of learning rate (=0.5), high value of discount factor (= 0.96) and low value of exploration decay rate (=0.15) perform well as the number of wins of our Q-Agent saturates early and performs better continuously. All graphs show the trend that the Q-Agent is performing well against the Random agent which makes vague random moves so this trend of probability of wins saturates.

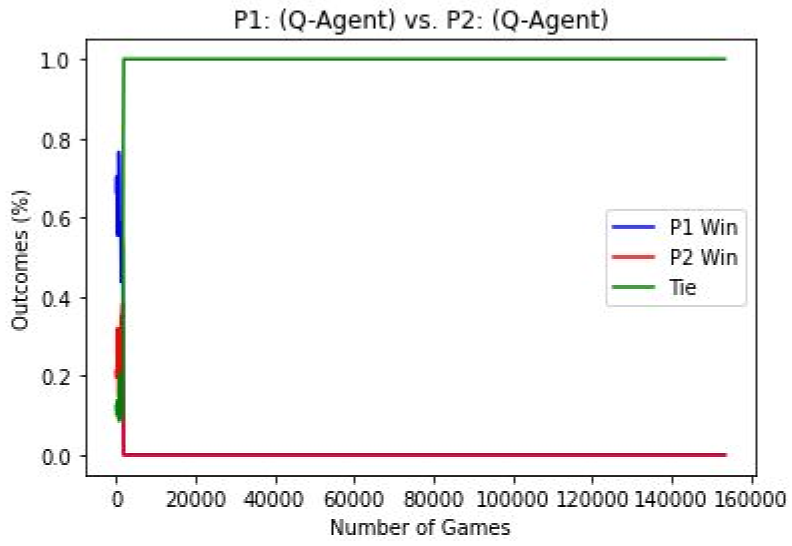
- **Q2: Q-Agent vs Q-Agent**



α = 0.9, ᵞ = 0.96, exploration decay rate = 0.01



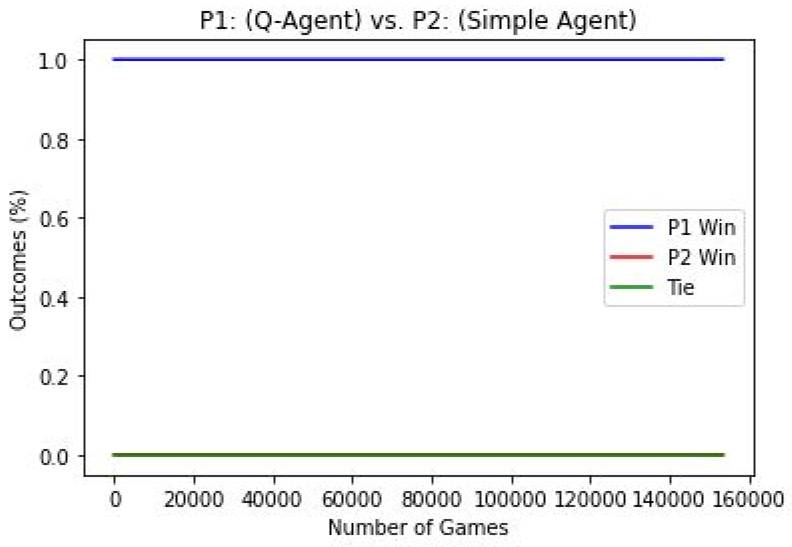
α = 0.7, ᵞ = 0.85, exploration decay rate = 0.05



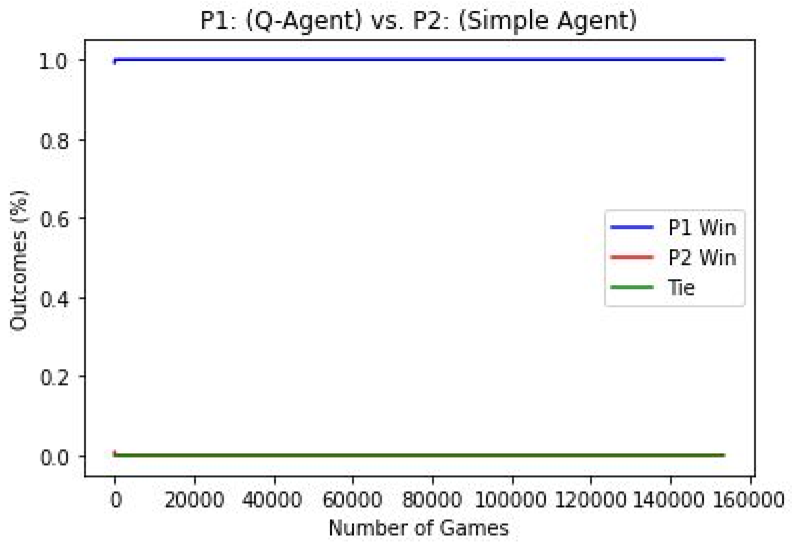
α = 0.5, ᵞ = 0.71, exploration decay rate = 0.15

All the graphs show the same trend that the number of wins of Q-Agents continuously decreases and probability of ties reaches the saturation i.e. 1, which is intuitive because both the agents are new agents so they learn at the same rate, hence, the number of ties continuously increases and saturates. From the three graphs, it is clear that large value of discount factor (=0.96), small value of learning rate (= 0.5) and small value of exploration decay rate (= 0.01) gives us better results. As the time taken by these Agents is higher to learn and hence we get better learned agents.

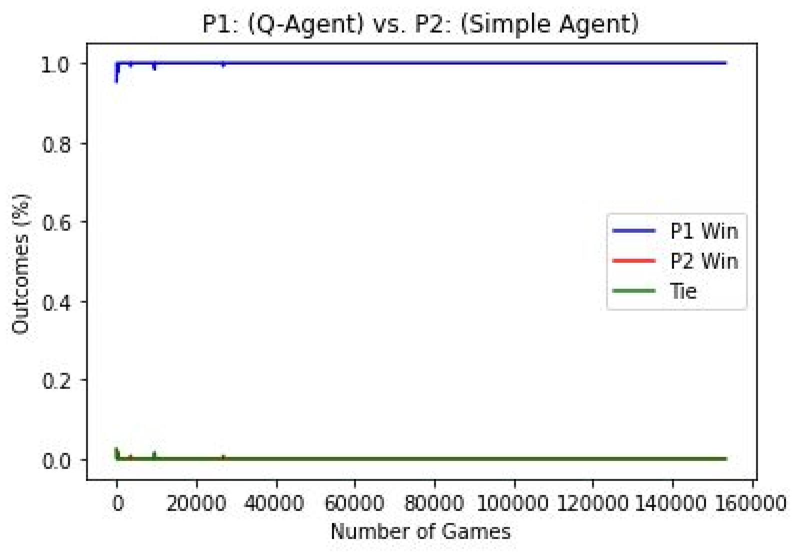
- **Q3: Q-Agent vs Simple Agent**



α = 0.9, ᵞ = 0.96, exploration decay rate = 0.01



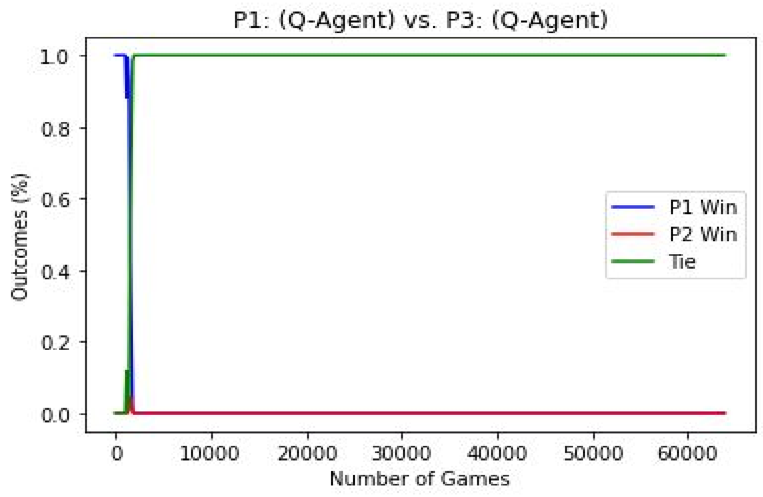
α = 0.7, ᵞ = 0.85, exploration decay rate = 0.05



α = 0.5, ᵞ = 0.71, exploration decay rate = 0.15

Training against Simple Agent is giving us very poor results as for this simple game, the Q-Agent without learning much performs very well almost always. Hence, the Agent obtained after learning would perform well in simple situations but not always.

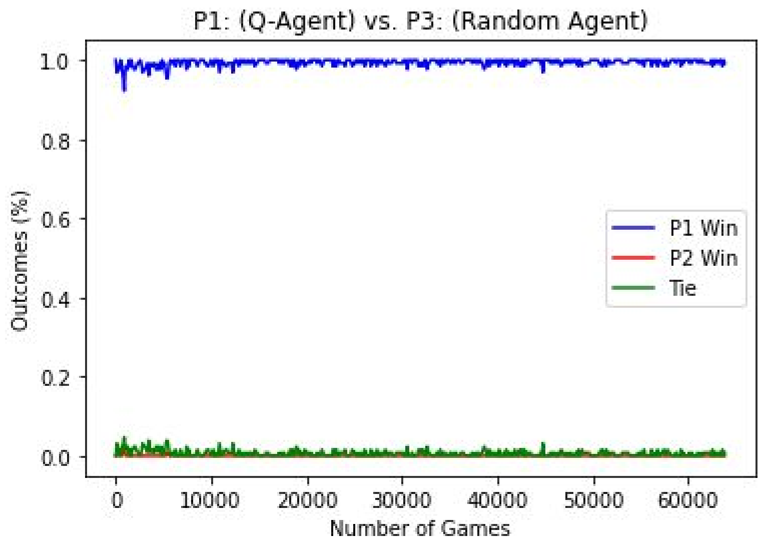
- **Q4: Q1 vs Q-Agent**



α = 0.9, ᵞ = 0.96, exploration decay rate = 0.01

The Q4 agent is obtained by training Q1 agent, obtained by training Q-Agent and Random Agent, against new Q-Agent. Results obtained are quite interesting. Initially Q1 agent is winning but after about 100 to 200 iterations, the number of ties increases. That implies that the new Q-Agent does not take much time to learn to the level of Q1.

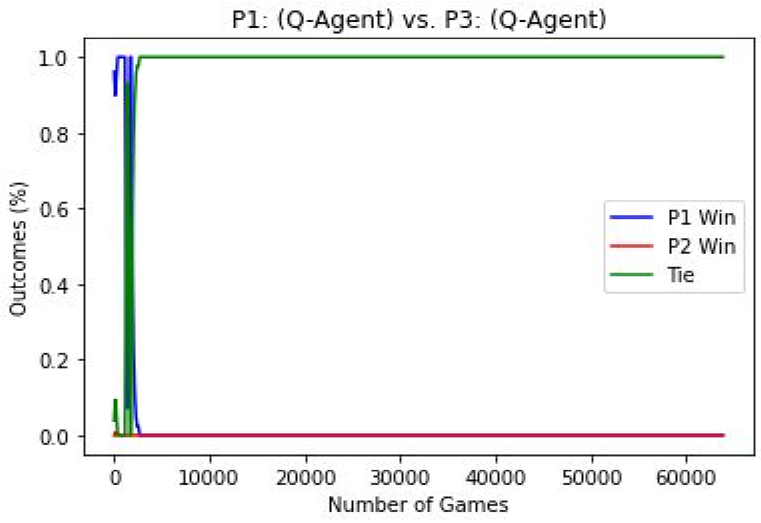
- **Q5: Q2 vs Random Agent**



α = 0.9, ᵞ = 0.96, exploration decay rate = 0.01

Well the results are not at all surprising as the Q2 agent which is already well trained is performing quite well with almost winning all the games. Initially the number of ties is much higher, but our Q-Agent adapts to the random agent quite well.

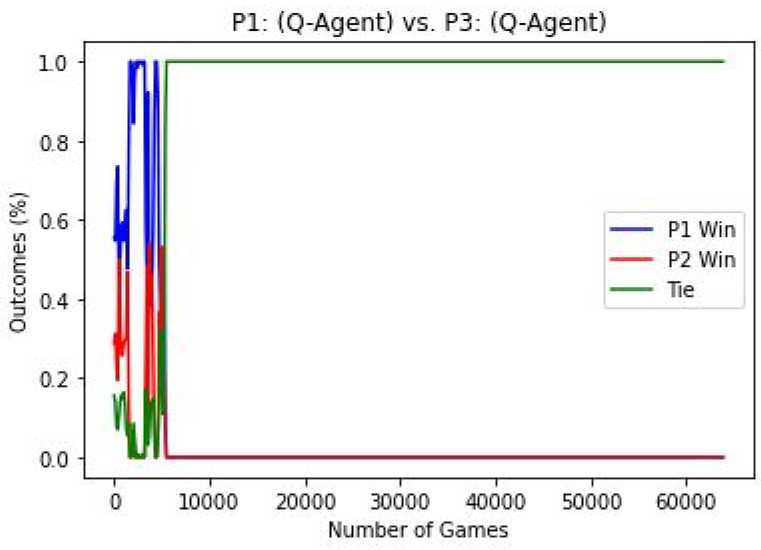
- **Q6: Q4 vs Q5**



α = 0.9, ᵞ = 0.96, exploration decay rate = 0.01

Q4 and Q5, both the agents are well trained agents so the results obtained are quite interesting. Q4 agent is performing quite well initially but after about 4000 games, both the agents have learned well enough that further games start to tie. Hence, the probability of ties gets close to one.

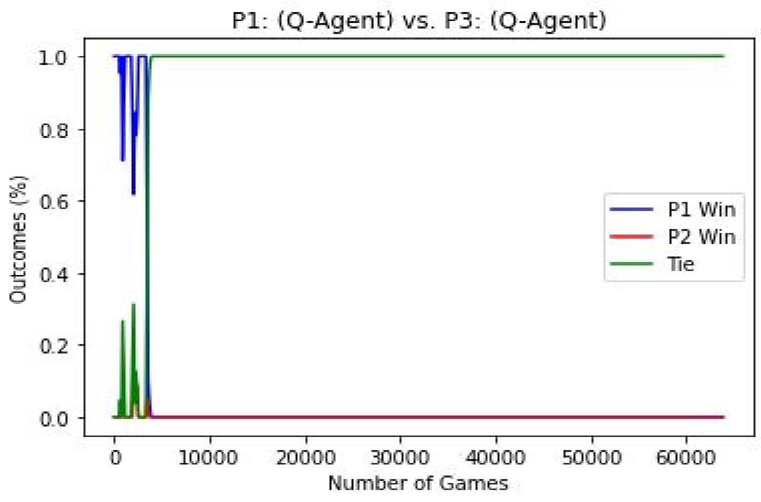
- **Q7: Q3 vs Q1**



α = 0.9, ϒ = 0.96, exploration decay rate = 0.01

As our test before showed that Q-Agent trained against Simple Agent is quite a weak trained agent, hence the results obtained are not much surprising. Q1 Agent is performing much better initially but after about 5000 games the results saturate to a probability of ties to 1.

- **Q8: Q1 vs Q2**



α = 0.9, ϒ = 0.96, exploration decay rate = 0.01

Both Q1 and Q2 are well trained agents. Q2 being trained against Q-Agent performs considerably good for initial 250 iterations but afterwards the results satutates again as other agents learn to tie. Hence, the probability of ties saturates to 1.

4. Conclusion

From all the tests performed, a few conclusions can be drawn. First, a higher discount factor always seems to give better results, independent of opponent or learning rate. In fact it is probably a good idea to look at even higher discount factors when trying to implement a game like Tic-Tac-Toe. Secondly, for a good end result a high learning rate is probably a good idea. Q-learning algorithms start to give good results even at low numbers of iterations for small games like this. Agents give fairly good results even for 20000-30000 games. Even for Simple Agent vs Q-Agent only took only 1000 to 2000 iterations for $\alpha = 0.5$, $\gamma = 0.71$, exploration decay rate = 0.15 and only 200 for $\alpha = 0.9$, $\gamma = 0.96$, exploration decay rate = 0.01. Our analysis further gives fair indication that data that lower the value of exploration decay rate give further improved results. Our analysis gives best results for $\alpha = 0.9$, $\gamma = 0.96$ and exploration decay rate = 0.01.

It is more or less intuitive to think that Q1, with experience against the Random agent, shows excellent results against Q3, with experience against the Simple Agent. Q1 is still not as good as Q2, which is trained against another Q-learning agent. Likewise, the new Q-learning agent has no problem overcoming the already experienced Q1 for a while. This gives among the first three agents Q1, Q2, and Q3, Q2 which is trained against the Q-Agent performing the best. Q1 is the second best Agent which is trained against Random agents as the graphs which compare against Q1 shows. Q3 performs pretty poorly as the graph shows that it takes only 4 to 5 games to learn which is quite low. Among Q4 and Q5, Q4 performs quite well.

5. Future Work

In the future, there will be agents with deep Q-learning, a convolutional neural network, and policy gradient descent. Learning to play these types of games is a great exercise in reinforcement learning.

References

Sutton, Richard S., & Barto, Andrew G. 1998. Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA.

Deeplizard Reinforcement Learning - Goal Oriented Intelligence
https://deeplizard.com/learn/playlist/PLZbbT5o_s2xoWNVdDudn51XM8lOuZ_Njv

Wikipedia Q-learning https://en.wikipedia.org/wiki/Q-learning