Reinforcement Learning Applied to Pivit

Bernardo Ramalho, *Master in Informatics and Computing Engineering, FEUP*
and
Pedro Pereira, *Master in Informatics and Computing Engineering, FEUP*

Abstract

Proposed by our teachers, this paper seeks to train an AI using Reinforcement Learning algorithms. We will use SARSA and Q-Learning algorithms in conjunction with an OpenAI Gym environment made in python. We will use those algorithms to teach two AIs how to play Pivit and try to unravel which is the best.

Index Terms

Reinforcement Learning, Pivit, Q-Learning, SARSA, Machine Learning, Minimax, OpenAI, Gym, Python, Pygame

Contents

I. Introduction

REinforcement learning is the basis for modern development of Artificial Intelligence that can outplay the top human players by teaching itself how to play. In this paper we will train an Agent to play Pivit using Q-Learning and SARSA algorithms. Our goal to determine which one gives us the best results with the same amount of training. We will conclude which one performs better by comparing the time they take to train, how much do they win while training and who plays better by playing them against each other. We will also try to explore some improvements to the algorithms. We hope this paper will bring more clarity regarding how Reinforcement learning works and when to use each of the algorithms.

August 27, 2020

II. Problem Description

Pivit is a boardgame created by Tyler Neylon in 2013. It can be played with 2 or 4 players but for this paper we chose to play with only 2 players. In this version of the game, each player has 12 pieces that are placed in an 8x8 board.

Rules:

- Each piece can only move in on direction (shown by the arrows drawn in it);
- Each piece can only move to a square of a different colour (if a piece is in a white square it can only move to a black one and vice-versa);
- When a piece moves it rotates 90^{o};
- When a piece reaches a corner of the board it evolves;
- An evolved piece can move to a square of the same colour;
- The game ends when there are only evolved pieces on the board.
- The winner is the player with the most pieces evolved pieces.

In order to train the Agent we will need to implement our own OpenAI Gym environment. This environment emulates the board and all its piece letting the agent play and receive rewards based on how well it performed. The environment should be able to calculate all the possible moves that can be made in a specific game state, calculate the reward of a specific game state and change the game state based on the action that the agent chooses to do.

In both SARSA and Q-Learning, the AI will play the game thousands of times against a random bot in order to learn with its mistakes. It will primarily take random moves but sometimes it will choose to take some move he already computed in a previous game. The information about the reward he gained with a specific action in a specific game state will be saved into a Q-Table. This Q-Table will then be used to choose the best action in a specific game state by looking at the reward that he has gained in previous games. By playing the game thousands of times, the Q-Table values (q-values) will be update , this way, the AI will know what's the best thing to do in any given time (assuming it has enought training). Since our game is way to complex to construct a table we will have to come up with a way to reduce the number of possible states without compromising the AI efficiency and power.

The difference between SARSA and Q-Learning is the way we update the value for a specific state and action in the Q-Table. The difference can be seen bellow:

- **Q-Learning:** $Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \times max_a(Q(s_{t+1}, a)) - Q(s_t, a_t))$
- **SARSA:** $Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \times Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$

This formulas are applied at every time an agent interacts with the environment. The terms used above represent:

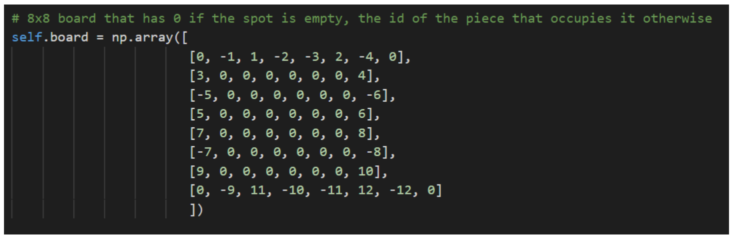
- $s_t \rightarrow$ Game State at time step t(a time step is a measure of time, it increments when the agent interacts with the world).
- $a_t \rightarrow$ Action taken by the agent at time step t.
- $\alpha \rightarrow$ Learning rate.
- $\gamma \rightarrow$ Discount factor for future rewards.
- $max_a(Q(s_{t+1}, a)) \rightarrow$ Maximum q-value for a game state at time step t + 1.

III. Approach

A. Gym Environment

1) Game Logic:

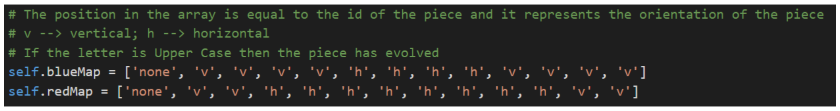
In our implementation of the game as Gym environment we use an 8 by 8 numpy 2-dimensional array (Figure 1). In each position $[x, y]$ with $x, y \in [0, 8[$ of the array we can have 0, if that position is empty, or a number representing the id of the piece that occupies the square. The ids of the pieces go from -12 to 12, excluding 0. The positive ids represent the pieces of the player that plays first (in our code, represented by the colour red) and the negative ones the pieces of the player that goes second (in our code, represented by the colour blue).



```
# 8x8 board that has 0 if the spot is empty, the id of the piece that occupies it otherwise
self.board = np.array([
                        [0, -1, 1, -2, -3, 2, -4, 0],
                        [3, 0, 0, 0, 0, 0, 0, 4],
                        [-5, 0, 0, 0, 0, 0, 0, -6],
                        [5, 0, 0, 0, 0, 0, 0, 6],
                        [7, 0, 0, 0, 0, 0, 0, 8],
                        [-7, 0, 0, 0, 0, 0, 0, -8],
                        [9, 0, 0, 0, 0, 0, 0, 10],
                        [0, -9, 11, -10, -11, 12, -12, 0]
                        ])
```

Fig. 1: Representation of the board in our environment.

To represent the direction of a piece and if the piece is evolved we use two arrays (Figure 2) (that we will call redMap for the read pieces and blueMap for the blue pieces), with size 13, one for each set of pieces (blue and red). In each index of the array we save a character that represents the direction of the piece. We use ’v’ if the piece moves vertically or ’h’ if the piece moves horizontally. To represent an evolved piece, we use upper case instead of lower case, in other words, if, for example, the value at array[2] is ’V’ then the piece moves vertically and is evolved. To easy the access to the information the value at position 0 of the arrays is equal to ’none’. This way we can access the information of the red piece with id 2 by calling array[2]. We also use ’none’ to represent if a piece has been eaten. By using this representation, we can check the final condition by simply iterating over the two arrays and checking if there are no lower case letters.

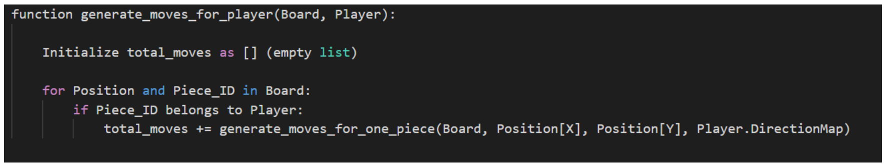


```
# The position in the array is equal to the id of the piece and it represents the orientation of the piece
# v --> vertical; h --> horizontal
# If the letter is Upper Case then the piece has evolved
self.blueMap = ['none', 'v', 'v', 'v', 'v', 'h', 'h', 'h', 'h', 'v', 'v', 'v', 'v']
self.redMap = ['none', 'v', 'v', 'h', 'h', 'h', 'h', 'h', 'h', 'h', 'h', 'v', 'v']
```

Fig. 2: Representation of the redMap and blueMap in our environment.

To define the logic of the game we created some auxiliary functions to generate all moves a player can make(Figure 3) by calling for every piece a function that generates all moves for that piece (Figure 4), to check if the game has ended(Figure) and a function to change the game state based on an action it receives (Figure 6).

To generate all moves a player can make, we call a function (Figure 3) based on the player turn. Both of those function do similar things but for different pieces (one for red pieces and other for blue pieces).



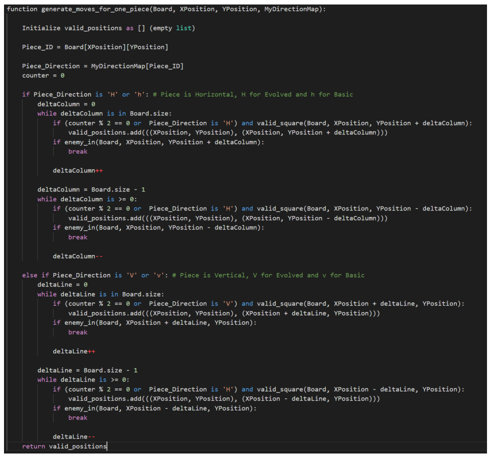
```
function generate_moves_for_player(Board, Player):

    Initialize total_moves as [] (empty list)

    for Position and Piece_ID in Board:
        if Piece_ID belongs to Player:
            total_moves += generate_moves_for_one_piece(Board, Position[X], Position[Y], Player.DirectionMap)
```

Fig. 3: Pseudo code of function that generates a player valid moves.

Those function run through the board, when then find a piece id that belong to them then they check the direction. Based on the direction we increment and decrement the position (in the x value if it is vertical, y value if it is horizontal) until we either find a valid square that has a piece in it or the edge of the board (Figure 4).

To check if the game has ended, we first check if a the current player can make a valid move (if it can’t the game ends). Then we iterate over the redMap and blueMap and if we find a lower case letter (either ’v’ or ’h’) then the game hasn’t ended. If we only find values ’none’ or upper case letter then the game is over.
In a gym environment we receive an action, that must be number, to change the game state. But to change the game state we need to use a dictionary called move. This dictionary has the following information:

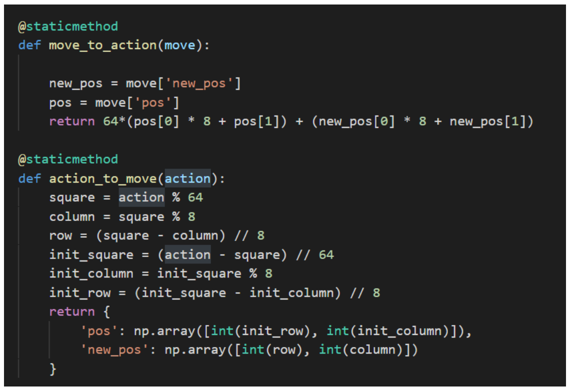
- Old position of the piece to be moves.
- The new position that the piece moved to.



```
function generate_moves_for_one_piece(Board, XPosition, YPosition, MyDirectionMap):

    Initialize valid_positions as [] (empty list)

    Piece_ID = Board[XPosition][YPosition]

    Piece_Direction = MyDirectionMap[Piece_ID]
    counter = 0

    if Piece_Direction is 'H' or 'h': # Piece is Horizontal, H for Evolved and h for Basic
        deltaColumn = 0
        while deltaColumn is in Board.size:
            if (counter % 2 == 0 or  Piece_Direction is 'H') and valid_square(Board, XPosition, YPosition + deltaColumn):
                valid_positions.add(((XPosition, YPosition), (XPosition, YPosition + deltaColumn)))
            if enemy_in(Board, XPosition, YPosition + deltaColumn):
                break

            deltaColumn++

        deltaColumn = Board.size - 1
        while deltaColumn is >= 0:
            if (counter % 2 == 0 or  Piece_Direction is 'H') and valid_square(Board, XPosition, YPosition - deltaColumn):
                valid_positions.add(((XPosition, YPosition), (XPosition, YPosition - deltaColumn)))
            if enemy_in(Board, XPosition, YPosition - deltaColumn):
                break

            deltaColumn--

    else if Piece_Direction is 'V' or 'v': # Piece is Vertical, V for Evolved and v for Basic
        deltaLine = 0
        while deltaLine is in Board.size:
            if (counter % 2 == 0 or  Piece_Direction is 'V') and valid_square(Board, XPosition + deltaLine, YPosition):
                valid_positions.add(((XPosition, YPosition), (XPosition + deltaLine, YPosition)))
            if enemy_in(Board, XPosition + deltaLine, YPosition):
                break

            deltaLine++

        deltaLine = Board.size - 1
        while deltaLine is >= 0:
            if (counter % 2 == 0 or  Piece_Direction is 'H') and valid_square(Board, XPosition - deltaLine, YPosition):
                valid_positions.add(((XPosition, YPosition), (XPosition - deltaLine, YPosition)))
            if enemy_in(Board, XPosition - deltaLine, YPosition):
                break

            deltaLine--
    return valid_positions
```

Fig. 4: Pseudo code of function that generates the valid moves for a certain piece.

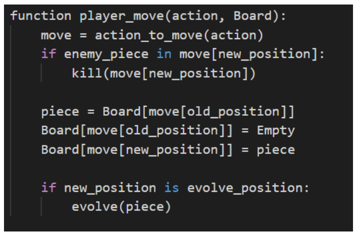
So we created two functions: one that receives an action and converts it into a move (we will call it actionToMove, Figure 5) and one that receives a move and converts it into an action (we will call it moveToAction, Figure 5) . These two functions are fundamental to help the environment communicate with the game logic.



```python
@staticmethod
def move_to_action(move):

    new_pos = move['new_pos']
    pos = move['pos']
    return 64*(pos[0] * 8 + pos[1]) + (new_pos[0] * 8 + new_pos[1])

@staticmethod
def action_to_move(action):
    square = action % 64
    column = square % 8
    row = (square - column) // 8
    init_square = (action - square) // 64
    init_column = init_square % 8
    init_row = (init_square - init_column) // 8
    return {
        'pos': np.array([int(init_row), int(init_column)]),
        'new_pos': np.array([int(row), int(column)])
    }
```

Fig. 5: Code of functions that convert action to move and move to action.

Lets focus on the actionToMove. To change the game state when we receive an action we use actionToMove. We call it to convert it to a move so we can change the game state according to the information saved there. First we check if there is a piece in the new position (if the move was generated then it can only be an enemy piece), if there is we kill it. Then we move the piece into the new position and check if it landed on a corner. In which case we evolve it.



```
function player_move(action, Board):
    move = action_to_move(action)
    if enemy_piece in move[new_position]:
        kill(move[new_position])

    piece = Board[move[old_position]]
    Board[move[old_position]] = Empty
    Board[move[new_position]] = piece

    if new_position is evolve_position:
        evolve(piece)
```

Fig. 6: Pseudo code of function that changes the game state based the action taken.

We this function (plus some function we don't see as relevant enough to talk in here) we cover all of the game logic.

2) *Gym Specific Functions*

Gym environments need a very specific organization to work. The key things are the function that we most create in our environment. They are:

- Setup: In this function we initialize everything that needs to be initialize to start a new game. We initialize the board, maps and the player turn.
- Step: In this function the environment receives an action and must act on it.
- Render: In this function we use pygame to display the game state on the screen.
- Reset: In this function we call setup to put the default values in all the variables.

The only function that it is worth talking about is Step since the other ones are pretty simple. In Step we start by checking if the received action is in the action space. Our action space goes from 0 to 64×65 because our moves have the current position and the old position (detailed information can be seen in Figure 5). If action is not in our action space then we throw an error. Else we use the function that changes the game state based on the action. We then check if the game has ended, if it has we return a reward based of who has won the game. If the game has not ended then we take the opponent action (in training the opponent takes a random move). We then check if the game as ended with the opponent action and calculate the reward depending on that. Then we return the reward and a Boolean telling if the game in done or not.

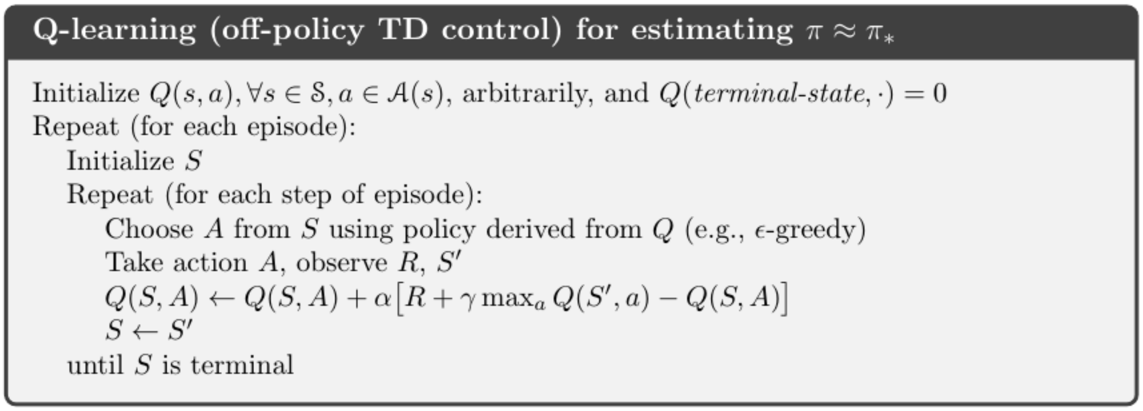
3) *Learning Algorithms*

The greatest challenge we faced while developing the algorithms was that our observation space was to large to start building a q-table from the start. To combat that we decided that instead of a traditional table we would use an hash table. The key values of this hash table would be the state itself. To do that we had to convert the state into a string we did that by creating a string that begins with the player turn (1 or -1) folllowed by the values that are on the board starting from the position [0,0] to position [7,7] (incrementing first the y value and when y reaches 7 we increment x by 1 and y comes back to 0). If we find a valid piece id (non zero) we convert it in a way that we can represent if the piece is an enemy or a ally, if the piece is vertical or horizontal and if the piece is evolved. We use 'e' to represent if it's an enemy piece and 'f' to represent an ally piece. These letters are followed by 'v' if the piece direction is vertical and 'h' if the piece direction is horizontal. To show if the piece is evolved we use 'V' or 'H' when we would use 'v' and 'h' accordingly.
This approach has some advantages:

- We have a reduced number of states since we don't care about individual pieces nor do we care about their colour.
- The AI is more versatile since it can play either with blue pieces or red pieces (if we hashed the state having to account the colour of the piece, if he played with different colour those would be different states).

We can get away with this because, in Pivit, the pieces are all the same. A player only needs to care about what pieces are theirs and what pieces are not. This also makes training the AI against itself (in Q-Learning) speed up training. Since the state doesn't care about which colour the pieces are, a state where we have a red pieces and blue pieces is equal to a state where we have them inverted (this is, the blue pieces are in the same position with the same direction as the red pieces and vice-versa).
The value of a certain key is another hash table. This hash table uses the action as the key and the value is the q-value of the state-action pair.

As for the implementation itself, a part from that and some slight modification we had to make to accommodate our unusual Q-Table, we use the algorithms shown in the Figures 7 and 8 for Q-Learning and SARSA, respectively.



```
Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize $S$
    Repeat (for each step of episode):
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
        $S \leftarrow S'$
    until $S$ is terminal
```

Fig. 7: Pseudo code of Q-Learning algorithm.

```
Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize $S$
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
    Repeat (for each step of episode):
        Take action $A$, observe $R$, $S'$
        Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
        $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
        $S \leftarrow S'$; $A \leftarrow A'$;
    until $S$ is terminal
```

Fig. 8: Pseudo code of SARSA algorithm.

IV. Experimental evaluation

To test our AI we ran 20000 episodes with the same parameters on both algorithms.

A. *Time Efficiency*

In terms of efficiency we thought that the Q-Learning would be the most efficient and that's what the tests show.
To test the time efficiency we calculate how long the AI took to complete the 20.000 episodes. In the graph (Figure 9), we can see that there is only a slight difference, approximately 2 minutes and 20 seconds (140 seconds). Although it doesn't seems much we think that if we had done more episodes the difference would only grown . In terms of efficiency of training, we concluded that the Q-Learning algorithm is the best.

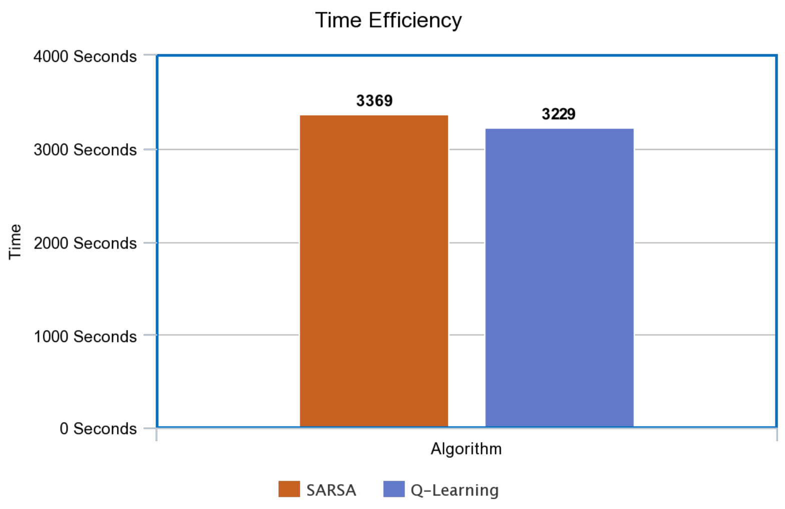


Fig. 9: Time Efficiency.

B. Win Rate Against Random Bot

We had predicted that the Q-Learning would win way more against the bots but that was not what the test show. If the AI stopped because it surpassed the amount of maximum plays we considered that as a draw (giving reward 0).
In terms of raw power, both of the algorithms show similar win rates in training. If we look at the graph (Figure 10), we see that both of them have, approximately, 50 percent win rate. Q-Learning has a 0.36 percent more then its counter part which equal to 70 games in 20.000. We don't think is is a substantially increase in games won. This might do with they being somewhat similar or to the amount of episodes.

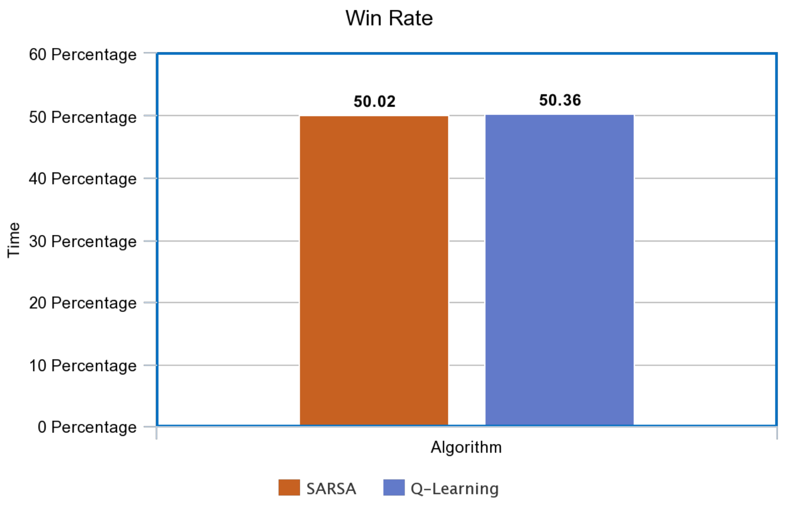


Fig. 10: AI vs Random Bot Win Rate graph.

C. Rewards Per Thousands Episodes

In terms of rewards per thousand episodes, we stored every reward the AI gained in training and then divided them into groups of 1000. We averaged the values in those groups and compared them.
To our surprise we see that the bots don't seem to be learning as well as we tought they would. As we can see by the graph (Figure 11), the average rewards don't go up (which was the expected), this means that they are not learning that well. We think this is due to the number of episodes we ran. We would like to retry this experiments with a way of running way more episodes but our computers simply could do it in a feasible time frame due to the delivery date. One positive thing we can see is that the consistency of the SARSA algirthm actually goes up as the episodes go by.

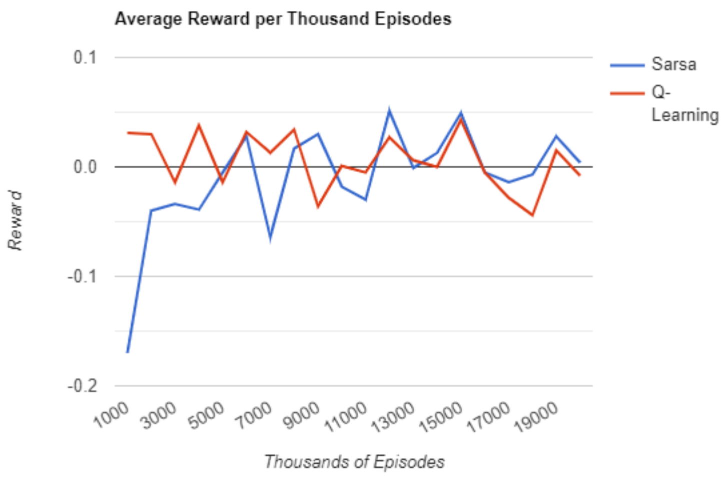


Fig. 11: Rewards per Thousands Episodes graph.

D. *Win Rate Against Each Other*

To test the win rate we made them play against each other 1000 times. We considered that if they reached 500 actions the game would be a draw. With that said, we thought the Q-Learning would win more times and that's exactly what happened. By looking at the graph (Figure 12), we can see that the AI that trained under Q-Learning won 15 more times then the one trained under SARSA. Although this might seem a low number we do think it is sufficient to conclude that the Q-Learning AI has a slight advantage to its SARSA counter part. Something that troubles us is the number of draws that they have but we can conclude that to be normal because, in Pivit, has the number of pieces starts to fall it becomes harder and harder to win (especially if both player have the same amount of pieces).

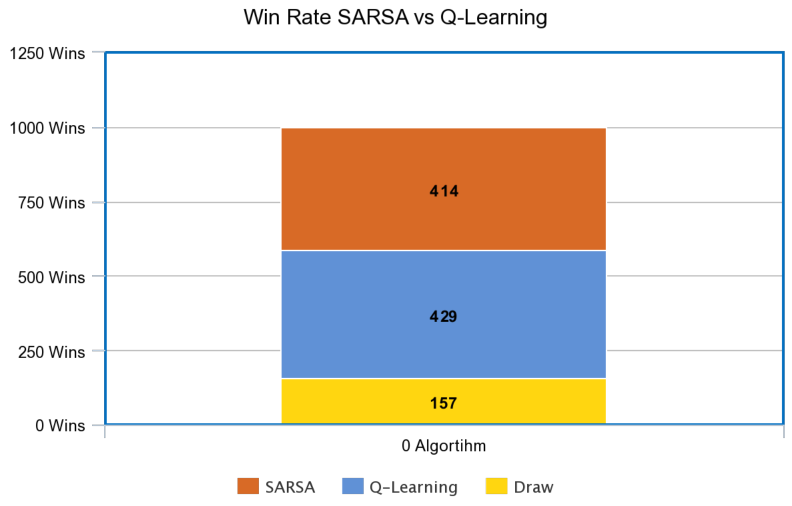


Fig. 12: SARSA vs Q-Learning Win Rategraph.

V. Conclusion

Based on our experimental evaluation we can conclude that the Q-Leaning AI is both better in terms of time efficiency and raw power. This is probably do to the the way it updates the q-values from the function. Although our rewards per thousand episodes where not what we expected, we don't think that influences the other results. Mainly because they both had the same training, in the same environment and in the same computer.

We also don't think there was any problem with the environment since we did it based on previous environment done be OpenAI and various other researches around the world. We deeply tested it to make sure nothing was failing. We tested move generation, move conversion, state changing, valid inputs and much more.

After analysing all the data that we gathered, we do think that it would have been better to train the AIs in more episodes (maybe 100.000 or 500.000) because although Pivit is a simple game on the surface, it has a large amount of possible states. We hope to actually trying this in the future since we didn't had time for this paper.

In continuation of this work we would like to implement a deep neural network as to improve the training as we could implement a deep Q-learning algorithm. This would vastly outperform our algorithm and give much better results.

All in all, we do realize that this research really helped us understand better the way Reinforcement Learning works and where to use it. Which is super important on this new age of AI thats starting to grow.
We also got the knowledge on how to make custom OpenAi Gym environment which will be helpful in our future experimentation with Reinforcement Learning.
We are very grateful for the opportunity that our faculty gave us of studying RL with some depth and we hope that we can in the future have more opportunities like this one.

References

[1] Reinforcement Learning Algorithms,
https://www.cse.unsw.edu.au/~cs9417ml/RL1/algorithms.html
[2] Q-Learning in Python,
https://www.geeksforgeeks.org/q-learning-in-python/

[3] Reinforcement Learning - Goal Oriented Intelligence Course,
https://deeplizard.com/learn/playlist/PLZbbT5o_s2xoWNVdDudn51XM8lOuZ_Njv
[4] UCL Course on RL,
https://www.davidsilver.uk/teaching/
[5] States, Observation and Action Spaces in Reinforcement Learning,
https://medium.com/swlh/states-observation-and-action-spaces-in-reinforcement-learning-569a30a8d2a1
[6] Temporal-Difference, SARSA, Q-Learning Expected SARSA in python, https://towardsdatascience.com/reinforcement-learning-temporal-difference-sarsa-q-learning-expected-sarsa-on-python-9fecfda7467e