# Reinforcement Learning applied to Pivit

Trabalho Realizado por:
Bernardo Ramalho - up201704334
Pedro Pereira - up201708807

# Especificação

Pivit é um jogo de tabuleiro criado em 2013. Tem as seguintes regras:

- Cada peça movimenta-se na direção indicada pelas setas que tem;
- As peças só se podem movimentar para quadrados de cores diferentes às que estão;
- Ao movimentar a peça roda 90º (mudando assim a direção das setas que tem);
- Uma peça pode ser promovida ao chegar a um espaço na ponta do tabuleiro;
- Uma peça promovida pode andar em qualquer direção;
- O jogo acaba quando não houver mais peça não promovidas em jogo e ganha quem tiver mais peças promovidas

Fig.1 - Tabuleiro do Pivit

# Pesquisa Realizada

- General Reinforcement Learning Algorithms Overview (Medium Article);
- Q Learning in Python (GeekforGeeks Article);
- Q Learning in Unity (Unity3D Blog Post);
- RL in Unity Tutorial (Unity Official Video);
- PyTorch Vs TensorFlow (Builtin Article);
- PyTorch Vs TensorFlow (Towards data science Article);
- Creating a Gym Environment (NovaTec Article);
- Reinforcement Q Learning from Scratch in Python with OpenAi Gym (Learndatasci Article);

# Ferramentas e Algoritmos

Para este trabalho iremos utilizar Python 3 com a framework Gym do OpenAI  para implementar os seguintes algoritmos:

- Q-Learning
- SAC (Soft Actor-Critic)

# Implementação Realizada

```python
# 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. 2 - Implementação do Tabuleiro

```python
# 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. 3 - Implementação do Estado de cada Peça

# Implementação Realizada

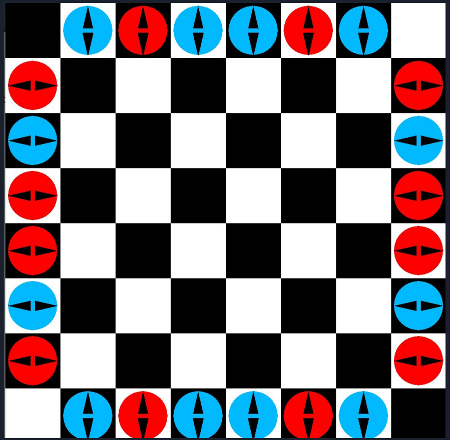

Fig. 4 - Interface Gráfica do Jogo.

```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 - Lógica de Conversão de Inteiro para estrutura de jogada

# Implementação Realizada

```
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. 6 - Lógica da geração de movimentos do jogo. (1 Peça)

```
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. 7 - Lógica da geração de movimentos do jogo.

```
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. 8 - Lógica da aplicação de jogada.

# Modelos implementados

**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)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\left[R + \gamma Q(S', A') - Q(S, A)\right]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until $S$ is terminal

Fig. 9 - Pseudo-código SARSA.

**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 $R$, $S'$
    $Q(S, A) \leftarrow Q(S, A) + \alpha\left[R + \gamma \max_a Q(S', a) - Q(S, A)\right]$
    $S \leftarrow S'$
  until $S$ is terminal

Fig. 10 - Pseudo-código Q-Learning.

# Dados Relevantes

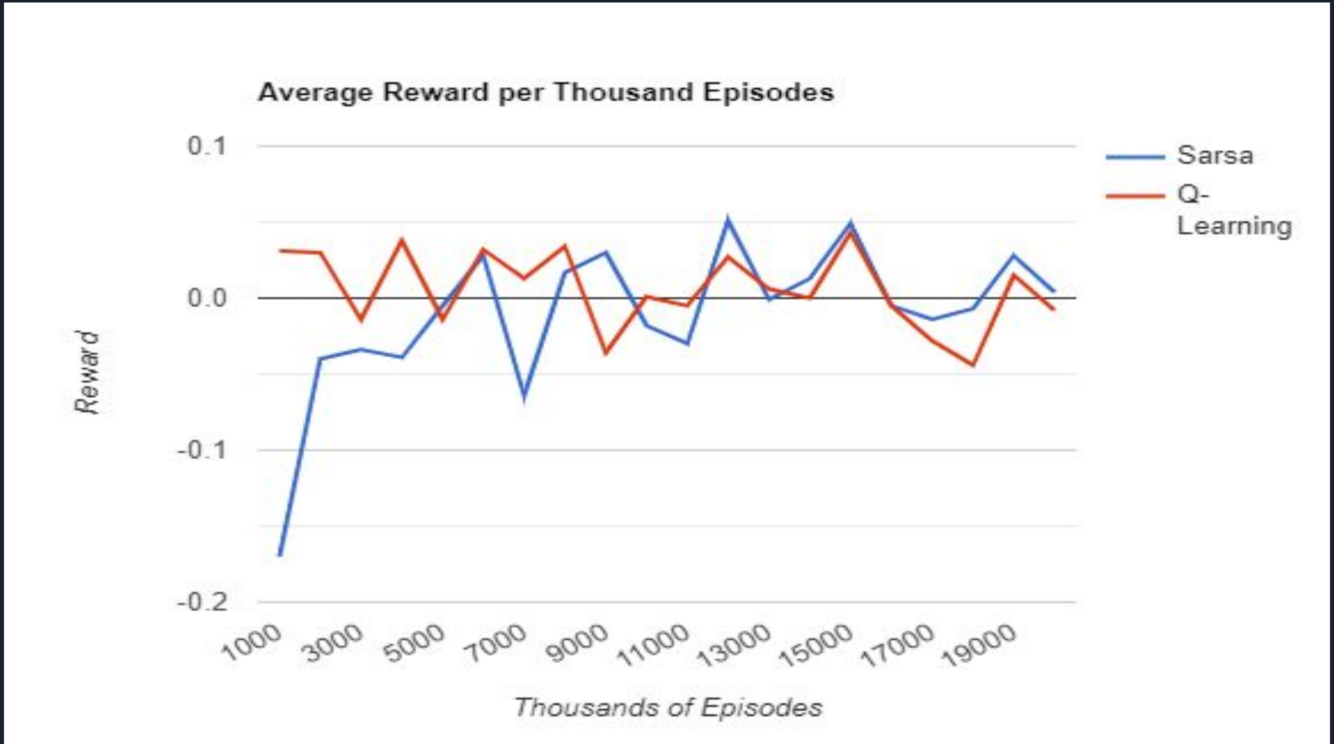


Fig. 11 - Recompensa Média / 1000 episódios

# Dados Relevantes

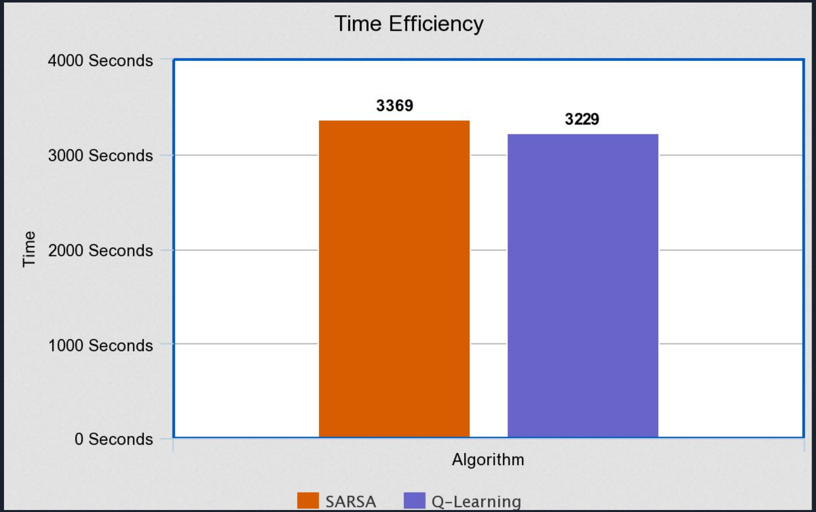


Fig. 12 - Tempo para correr 20000 episódios de treino em cada Alg.

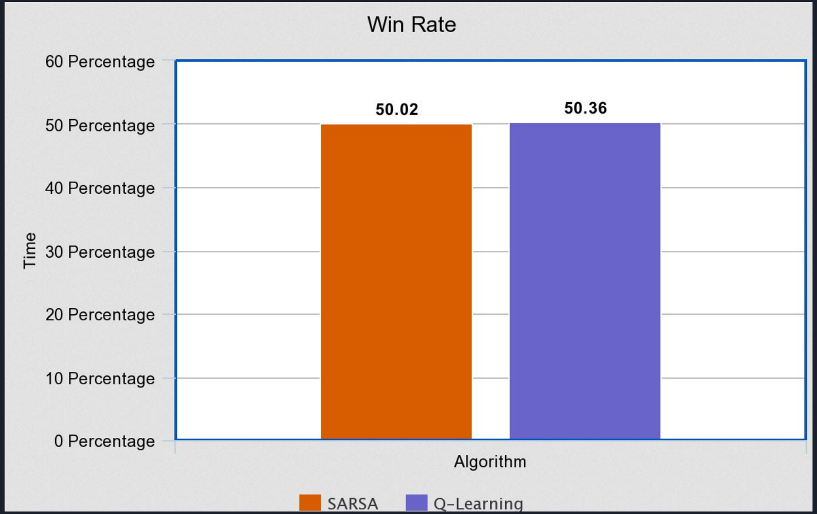


Fig. 13 - Taxa de vitória contra oponente de treino

# Dados Relevantes

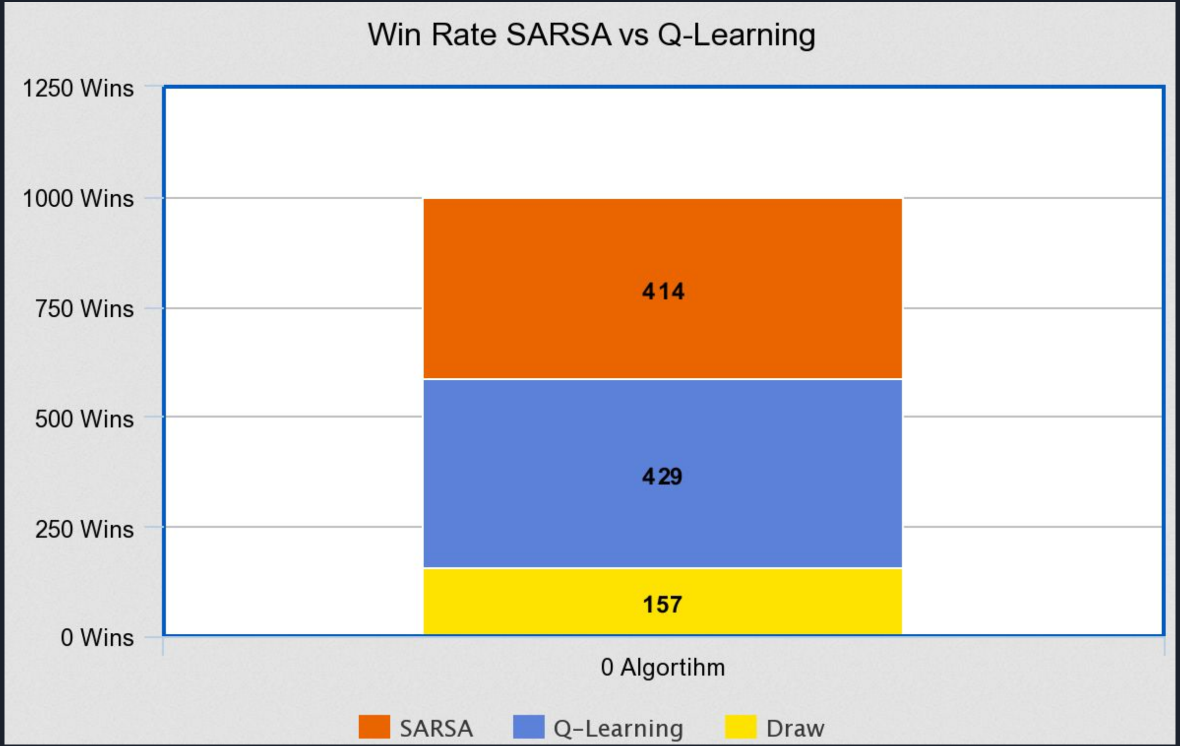


Fig. 11 - Resultados de 1000 episódios SARSA vs Q-Learning

FIM