MSBA – Optimization II
Project 3 – Reinforcement Learning (Connect Four PopOut)

Team Members: *Andrew Gillock (apg2255), April Kim (hk22849), Olivia Lee (rl29986), Parthiv Borgohain (pb25347)*

Problem Statement

In the gaming industry, developing games with more competitive opponents for single players is essential for gaming companies because it can lead to increased player satisfaction and retention, new revenue opportunities, replayability, and a positive brand image.

To find good strategies and actions in playing games, reinforcement learning, reinforcement learning, a type of machine learning that involves training an agent to learn from its environment through trial and error, can be implemented in a manner that the agent receives rewards or punishments based on its actions and tries to discover the best strategy to maximize its rewards.

In this project, we explore reinforcement learning on a simple game called PopOut, a variation of a well-known game Connect Four. The fundamental concept of PopOut involves two options for a player during their turn. They can either drop a checker into a column with empty spaces, as in the standard Connect Four game, or remove one of their checkers from the bottom row, provided they have one there. If a player opts to remove their checker from the bottom row, the checkers above it will shift down by one row. Removing one of the opponent's checkers on the bottom row is not permissible.

This report will prove the effectiveness of reinforcement learning as a powerful tool for finding optimal game strategies and thus help big business decisions on hiring reinforcement learning experts to work at our company.

Connect Four PopOut: The Game

Connect Four PopOut is a variation of the classic Connect Four game, which was invented by Howard Wexler in 1974 and published by Milton Bradley (now Hasbro). The game gained popularity quickly due to its simple rules and strategic gameplay. In Connect Four PopOut, players have the option to not only drop checkers into columns like in the standard version of Connect Four, but also to remove their own checkers from the bottom row, causing the checkers above to drop down. This adds an additional layer of complexity to the game and requires players to carefully consider their moves in order to win.

Strategies

1. Column Control: Controlling the columns is crucial in Connect Four PopOut. By strategically placing checkers in columns, players can block their opponent's moves and create opportunities for their own checkers to connect vertically, horizontally, or diagonally.
2. Defensive Moves: Players should also be mindful of their opponent's moves and try to anticipate their strategies. Removing a checker from the bottom row can be a defensive move to prevent the opponent from making a winning move in the next turn.
3. Offensive Moves: Players should aim to create their own winning opportunities by planning ahead and setting up their checkers to connect in a winning pattern.
4. Balancing Column Control and Checker Removal: Players need to find the right balance between controlling the columns and using the checker removal option strategically. Removing a checker from the bottom row can create new possibilities for connecting checkers, but it also reduces the overall number of checkers on the board, potentially giving an advantage to the opponent.

Winning Conditions

The objective of Connect Four PopOut is to be the first player to connect four of their own checkers in a vertical, horizontal, or diagonal line on the game board. Once a player achieves this, they win the game. The game can also end in a draw if the entire game board is filled with checkers and no player has achieved a winning pattern.

Rules

1. Players take turns dropping their own colored checkers into columns that have open spaces or removing their own checkers from the bottom row.
2. Players cannot drop a checker into a column that is already full, and they cannot remove a checker from the bottom row if it is not their color.
3. Checkers dropped into columns will stack on top of each other, and the checkers above the removed checker from the bottom row will drop down.
4. The first player to connect four of their own checkers in a vertical, horizontal, or diagonal line wins the game.
5. If the entire game board is filled with checkers and no player has achieved a winning pattern, the game ends in a draw.

In conclusion, Connect Four PopOut is a challenging and strategic variation of the classic Connect Four game. Players must carefully consider their moves, balance offensive and defensive strategies, and strategically use the option to remove checkers from the bottom row to gain an advantage over their opponent. Through reinforcement learning, we aim to develop agents that can effectively play Connect Four PopOut and outperform opponents using strategic gameplay and decision-making.

Methodology

I. Strategy: Policy Gradient Method

To begin this project, our team had to decide between two popular methodologies for RL: Q-Learning and Policy Gradients. The characteristic of Q-Learning is it approximates the value function using a neural network, and the input of the neural network is the state, and the output is the value function that corresponds to each possible action. The output layer of the neural network has k nodes if there are k possible actions. To choose the best action, the action with the highest value function is selected using an ϵ-greedy approach, balancing exploration and exploitation. Differently, Policy Gradients change the way neural networks are used in reinforcement learning. Rather than explicitly optimizing the value function, policy gradients pose the problem of picking the optimal action as a classification problem. A classification model is trained to take the state/frames as input and output the probability of each action being the best.

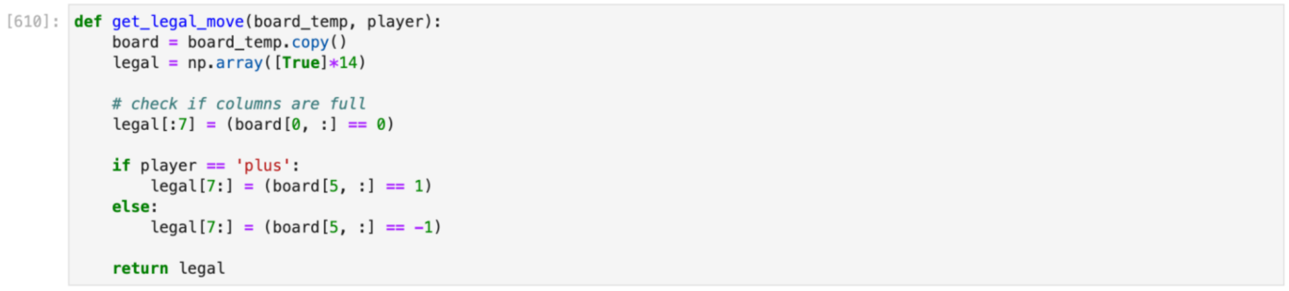
After exploring both characteristics of Q-Learning and Policy Gradients, our team decided to choose the Policy Gradient approach for our project. This is because Policy Gradients present the problem of picking the optimal action as a classification problem, where a classification model is trained to output the probability of each action being the best, without explicitly optimizing the value function. This method was more suitable for our project, which involved working with a complex and high-dimensional state space.

II. Functions used to set up Game

1. **update_board():** The "update_board" function in the project is responsible for taking the current board status, a color, and a column as input, and updating the board based on the specified move in the Connect Four PopOut game. It handles dropping a checker on the board if the column is not full, pulling a checker off the board if the column is in the valid range, and returns the updated board status.
2. **check_for_win():** The "check_for_win" function in the project is responsible for checking if there is a winner in the Connect Four PopOut game based on the current board status. It

iterates through the rows and columns of the board, checking for vertical, horizontal, and diagonal wins for both 'plus' and 'minus' players. If a winner is found, the function returns the type of win ('v-plus', 'v-minus', 'h-plus', 'h-minus', 'd-plus', or 'd-minus'), otherwise it returns 'nobody' indicating that there is no winner yet.

3. **display_board():** The "display_board" function in the project is responsible for displaying the current board status in ASCII format. It uses 'X' to represent '+1' and 'O' to represent '-1'. The function uses ASCII characters to draw the board with horizontal lines and vertical lines to separate the rows and columns. It also prints out the current state of the board with 'X' and 'O' representing the checkers of the players, and empty spaces for unfilled slots on the board.
4. **get_legal_move():** This function is responsible for determining the legal moves for the current player on the given board. It takes the current board state and the player ('plus' or 'minus') as input and returns a list of boolean values representing the legality of each move. The function checks if the columns are full and marks them as illegal if they are. It also checks if a piece can be popped out, based on the player's turn, and marks the corresponding moves as legal or illegal accordingly. The resulting list of boolean values represents the legal moves that the current player can make on the given board.



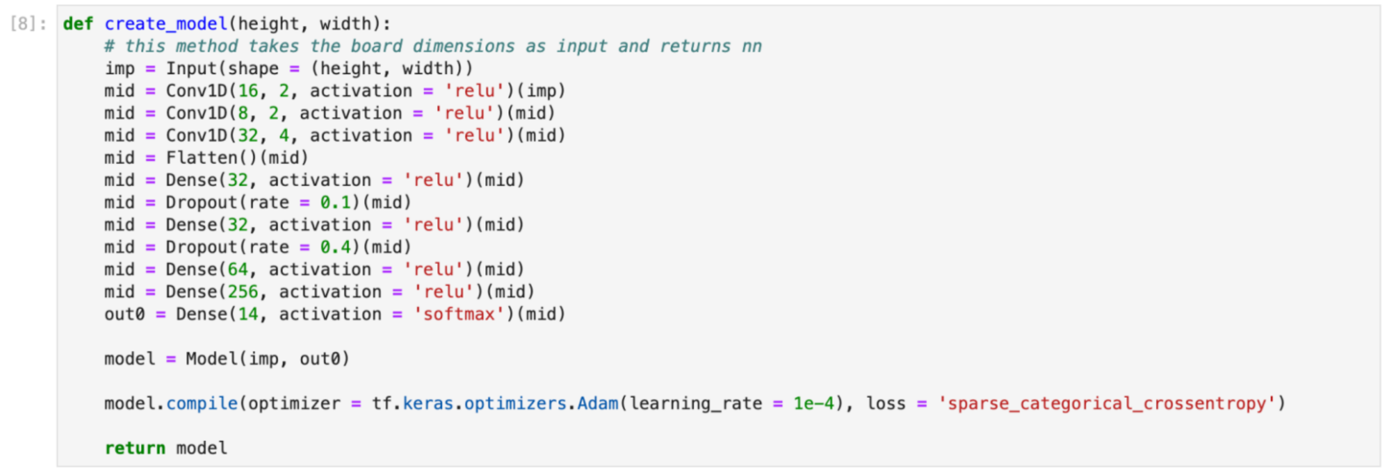
```
[610]: def get_legal_move(board_temp, player):
           board = board_temp.copy()
           legal = np.array([True]*14)

           # check if columns are full
           legal[:7] = (board[0, :] == 0)

           if player == 'plus':
               legal[7:] = (board[5, :] == 1)
           else:
               legal[7:] = (board[5, :] == -1)

           return legal
```

5. **play1random():** This function simulates a game of popout using a given model for the "plus" player and random moves for the "minus" player. It returns the winner of the game and arrays containing the sequence of boards, actions, rewards, and players for each move. The function uses epsilon-greedy exploration for the "plus" player, selects random moves for the "minus" player, updates the game state, checks for a winner, assigns rewards, and alternates player turns until a winner is determined.
6. **play1opp():** This function simulates a game of popout between two players, "plus" and "minus", using two different models, model1 and model2, for each player. The "plus" player selects moves based on probabilities obtained from model1 using epsilon-greedy exploration, while the "minus" player selects moves based on probabilities obtained from model2 assuming optimal behavior. The function updates the game state, checks for a winner, assigns rewards, and alternates player turns until a winner is determined. It returns

the winner of the game and arrays containing the sequence of boards, actions, rewards, and players for each move.

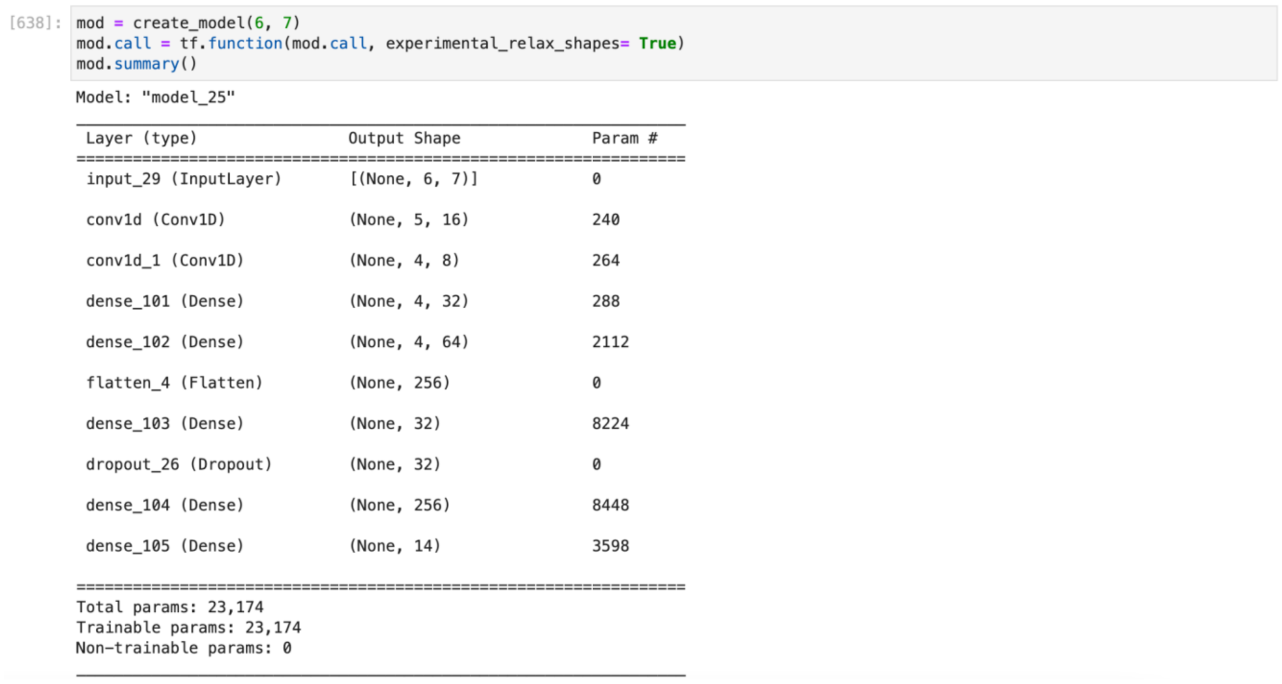
III. Create the Model

1. We created the function named create_model to create a neural network using Keras API to later train with our policy gradient.
2. The input layer has the shape of the height and width of the game board.
3. Then we added two 1D Convolutional layers using a ReLU Activation function with 16 and 8 filters with and a kernel size of 2, respectively.
4. Then, we add two fully connected hidden layers using a ReLU Activation function with 32 and 64 units and we add a line of code that flattens the output from the previous layer to a 1D array.
5. Again, we add another dense layer using a ReLU Activation function with 32 units, and add a dropout layer with a dropout rate of 0.1.
6. The next lines we add a fully connected layer with 256 units and a ReLU activation function.
7. Finally, we create the output layer with a softmax activation with 14 possible moves in the environment. The model then uses these 14 predictions to determine the best move to emerge victorious in the game.
8. Finally, we compile the model with RMSprop optimizer with learning rate of 1e-4 and use sparse categorical cross-entropy loss.



```
[8]: def create_model(height, width):
         # this method takes the board dimensions as input and returns nn
         imp = Input(shape = (height, width))
         mid = Conv1D(16, 2, activation = 'relu')(imp)
         mid = Conv1D(8, 2, activation = 'relu')(mid)
         mid = Conv1D(32, 4, activation = 'relu')(mid)
         mid = Flatten()(mid)
         mid = Dense(32, activation = 'relu')(mid)
         mid = Dropout(rate = 0.1)(mid)
         mid = Dense(32, activation = 'relu')(mid)
         mid = Dropout(rate = 0.4)(mid)
         mid = Dense(64, activation = 'relu')(mid)
         mid = Dense(256, activation = 'relu')(mid)
         out0 = Dense(14, activation = 'softmax')(mid)

         model = Model(imp, out0)

         model.compile(optimizer = tf.keras.optimizers.Adam(learning_rate = 1e-4), loss = 'sparse_categorical_crossentropy')

         return model
```

We called height of 6 and width of 7 in our create_model parameter, and we got summary of total parameters of 23,174.



```
[638]: mod = create_model(6, 7)
       mod.call = tf.function(mod.call, experimental_relax_shapes= True)
       mod.summary()
```

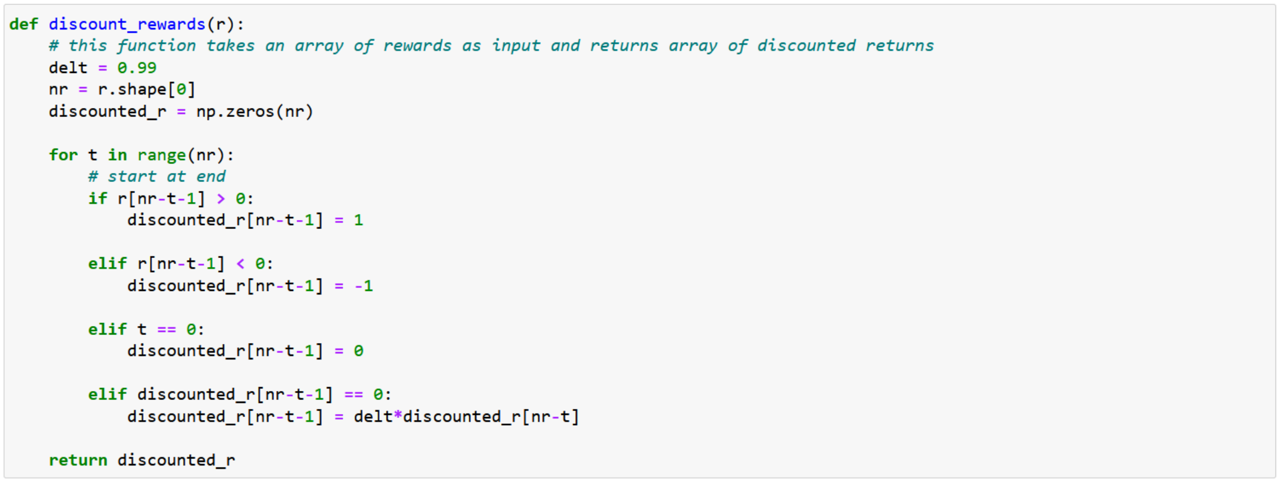
```
Model: "model_25"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_29 (InputLayer)       [(None, 6, 7)]            0

 conv1d (Conv1D)             (None, 5, 16)             240

 conv1d_1 (Conv1D)           (None, 4, 8)              264

 dense_101 (Dense)           (None, 4, 32)             288

 dense_102 (Dense)           (None, 4, 64)             2112

 flatten_4 (Flatten)         (None, 256)               0

 dense_103 (Dense)           (None, 32)                8224

 dropout_26 (Dropout)        (None, 32)                0

 dense_104 (Dense)           (None, 256)               8448

 dense_105 (Dense)           (None, 14)                3598

=================================================================
Total params: 23,174
Trainable params: 23,174
Non-trainable params: 0
_________________________________________________________________
```

IV. Process when a Game is Played

1. Initializes the game by creating an empty board size of 6 by 7 and epsilon value 0.05.
2. Until there is a winner, loop through the game and alternate between the two players.
3. For the player controlled by the model, the function calculates the probabilities of each legal move using the model
    a. Selects a move greedily based on the highest probability.
    b. Selects a move randomly with probability epsilon
4. The move is randomly selected from the legal moves for other players.
5. After each move a player makes, a function updates the board, determines the game's winner, and awards a reward based on the result.
6. The function also records the board state, action taken, reward received, and player who made the move, to use for later to train the model
7. The function will return the winner of the game, the recorded arrays of board states, actions, rewards, and players.

V. Discount Rewards Function

The "discount_rewards" function takes an array of rewards as input and calculates the discounted returns for each reward based on a discount factor (delt) of 0.99. It iterates through the rewards array in reverse order (starting from the end), and assigns discounted returns based on the following rules: if a reward is positive, the discounted return is set to 1; if a reward is negative, the discounted return is set to -1; if a reward is zero and it's the first reward, the discounted return is set to 0; otherwise, the discounted return is calculated as the product of delt and the previous discounted return. The function returns an array of discounted returns corresponding to the input rewards array.



```python
def discount_rewards(r):
    # this function takes an array of rewards as input and returns array of discounted returns
    delt = 0.99
    nr = r.shape[0]
    discounted_r = np.zeros(nr)

    for t in range(nr):
        # start at end
        if r[nr-t-1] > 0:
            discounted_r[nr-t-1] = 1

        elif r[nr-t-1] < 0:
            discounted_r[nr-t-1] = -1

        elif t == 0:
            discounted_r[nr-t-1] = 0

        elif discounted_r[nr-t-1] == 0:
            discounted_r[nr-t-1] = delt*discounted_r[nr-t]

    return discounted_r
```

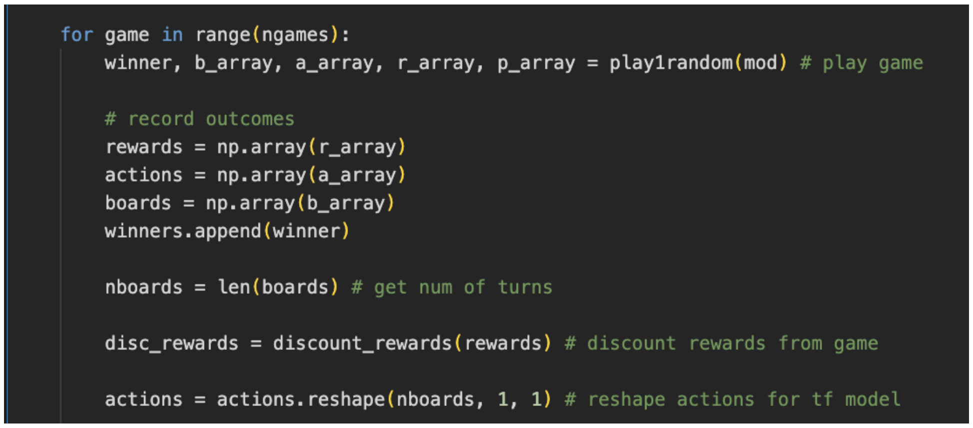
VI. Train

We decided to train our first model against an opponent who chooses all moves at random. This would provide our model with a wide variety of board arrangements and rewards to learn from. We train by utilizing the model.fit() method from tensorflow. Once 2000 games had been played, we saved this model and used it as the opponent for a new model. The model was evaluated using 500 games against a random opponent. The new model was trained for 10000 games against the first model. Finally, the new model was tested on the random opponent.

Results

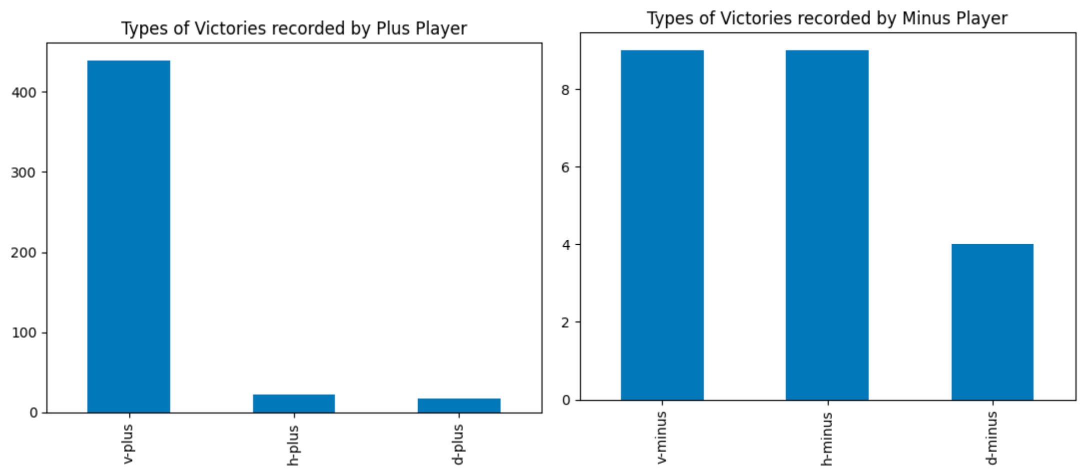
I. Model 1 vs Random Opponent

We used the following code to evaluate our model. After training the model for 2000 games, we simulated 500 games and recorded the results, which can be seen below.



```
for game in range(ngames):
    winner, b_array, a_array, r_array, p_array = play1random(mod) # play game

    # record outcomes
    rewards = np.array(r_array)
    actions = np.array(a_array)
    boards = np.array(b_array)
    winners.append(winner)

    nboards = len(boards) # get num of turns

    disc_rewards = discount_rewards(rewards) # discount rewards from game

    actions = actions.reshape(nboards, 1, 1) # reshape actions for tf model
```

```
For 500 games played, our model wins 95.6% of the time against an opponent picking randomly.
```



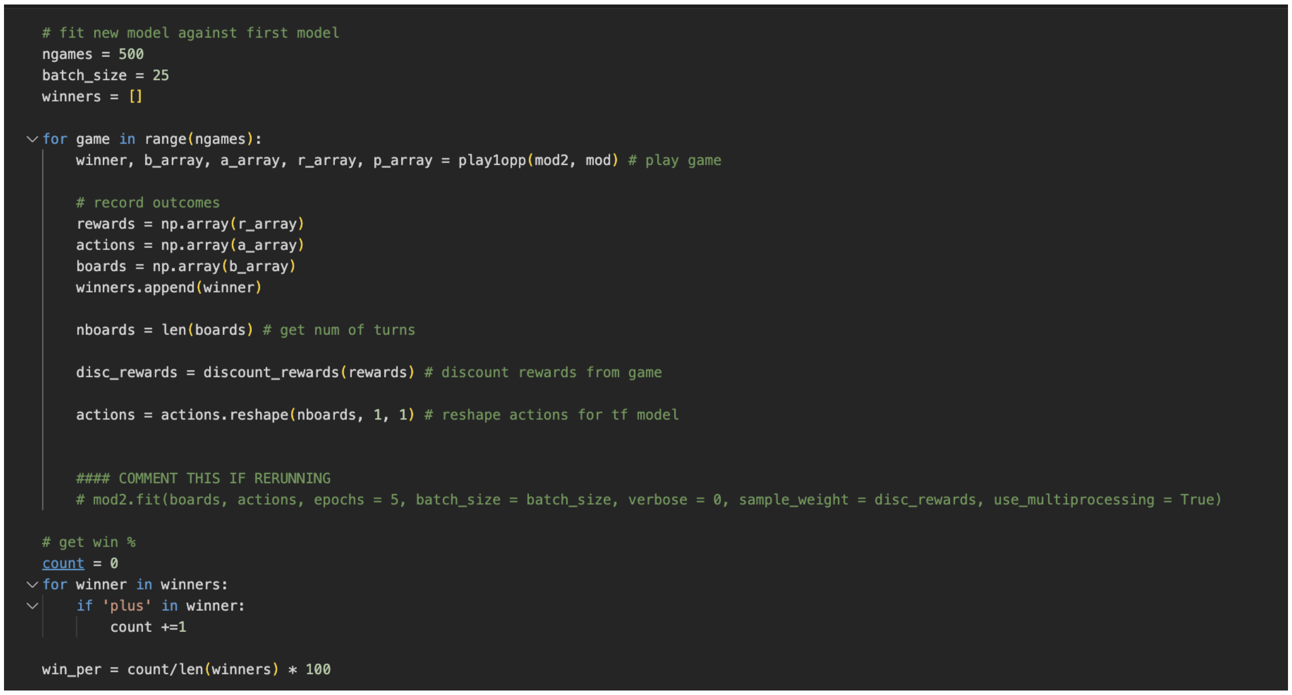
As we can see in the above plots, most of the games won by the Plus player are won vertically. A very small proportion of the victories of the Plus player are won in a horizontal manner. An even lesser proportion is won diagonally.

For victories recorded by the Minus Player, the proportion of vertical and horizontal victories is the same (9 victories each). A significant proportion of the victories are diagonal in nature too. For the minus player, the distribution of vertical,horizontal and diagonal victories is much more uniform than for the plus player.

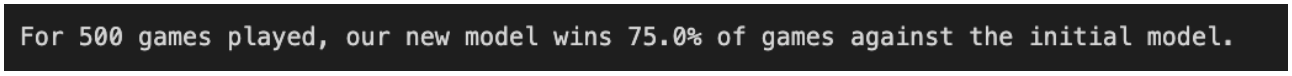
Our Model 1 won 95.6% of the 500 games against an opponent playing at random.

II. Model 2 vs Model 1

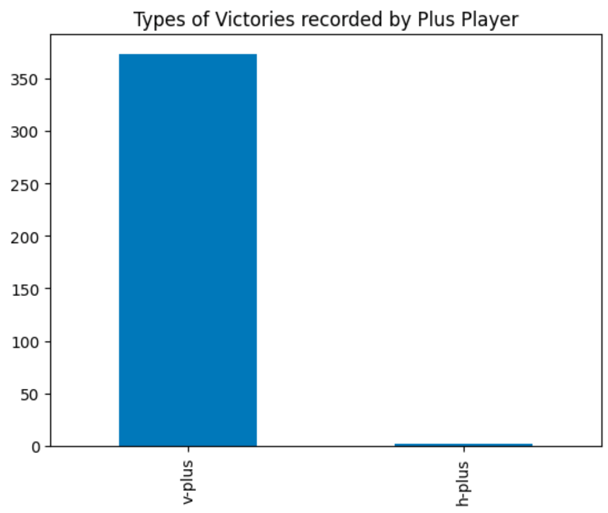
We trained a new model for 10000 games against the earlier model. We then let this new model play against the earlier model for 500 games and recorded the results.

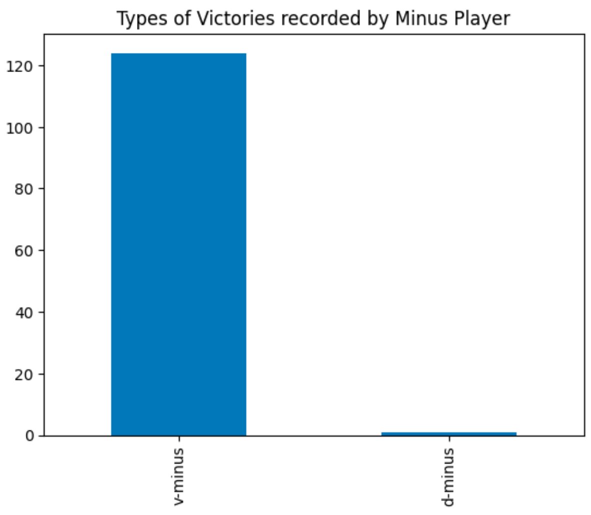


```
# fit new model against first model
ngames = 500
batch_size = 25
winners = []

for game in range(ngames):
    winner, b_array, a_array, r_array, p_array = play1opp(mod2, mod) # play game

    # record outcomes
    rewards = np.array(r_array)
    actions = np.array(a_array)
    boards = np.array(b_array)
    winners.append(winner)

    nboards = len(boards) # get num of turns

    disc_rewards = discount_rewards(rewards) # discount rewards from game

    actions = actions.reshape(nboards, 1, 1) # reshape actions for tf model


    #### COMMENT THIS IF RERUNNING
    # mod2.fit(boards, actions, epochs = 5, batch_size = batch_size, verbose = 0, sample_weight = disc_rewards, use_multiprocessing = True)

# get win %
count = 0
for winner in winners:
    if 'plus' in winner:
        count +=1

win_per = count/len(winners) * 100
```



```
For 500 games played, our new model wins 75.0% of games against the initial model.
```



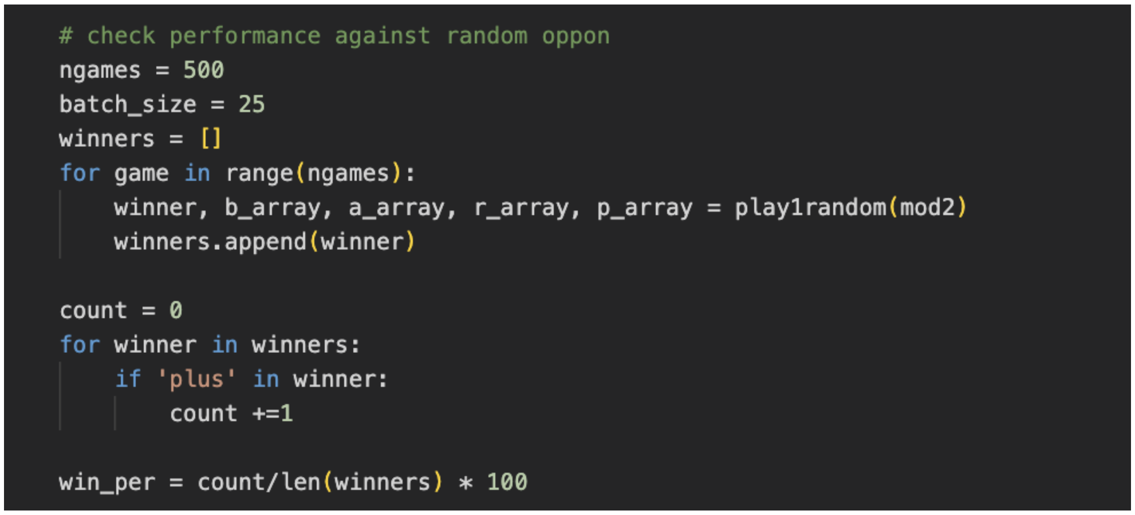


As seen in the above plots, when model 2 plays against model 1, nearly all of the victories recorded by the plus player as well as the minus player are vertical in nature.

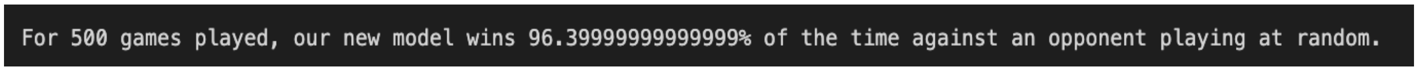
Our Model 2 won 75% of the 500 games played against our Model 1.

III. Model 2 vs Random Opponent

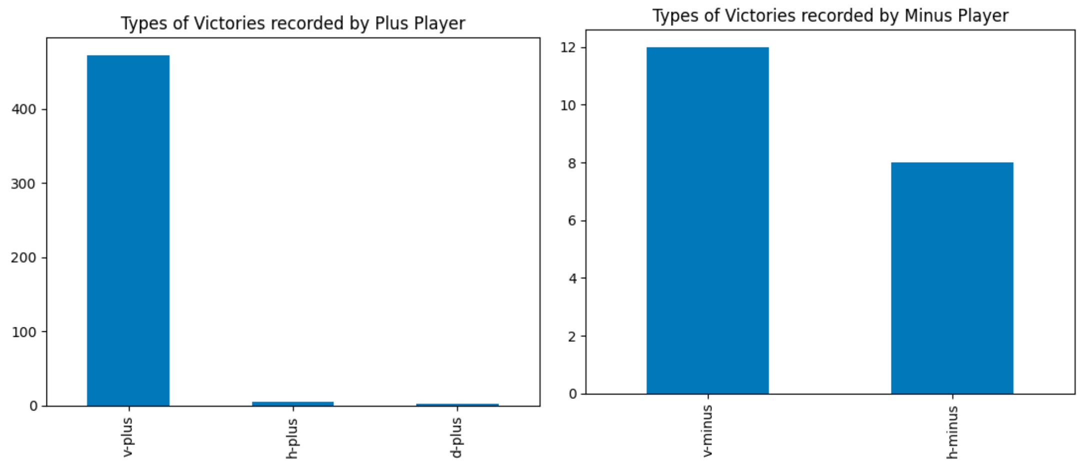
We then let the new model play against a random opponent for 500 games and recorded the results.



```
# check performance against random oppon
ngames = 500
batch_size = 25
winners = []
for game in range(ngames):
    winner, b_array, a_array, r_array, p_array = play1random(mod2)
    winners.append(winner)

count = 0
for winner in winners:
    if 'plus' in winner:
        count +=1

win_per = count/len(winners) * 100
```



```
For 500 games played, our new model wins 96.39999999999999% of the time against an opponent playing at random.
```



As we can see in the above plots, most of the games won by the Plus player are won vertically. A very small proportion of the victories of the Plus player are won in a horizontal manner. An even lesser proportion is won diagonally.

For the minus player, 12 victories were recorded in a vertical manner and about 8 victories in a horizontal manner. No victory was recorded diagonally.

Our Model 2 won 96.4% of the 500 games against an opponent playing at random. Overall, there was not a significant increase in accuracy despite the 10,000 training games. We believe that this is due to our network not getting enough exposure to different board types.

Limitations & Next Steps

The biggest limitation of our project is the time constraint. Having a time constraint limits the amount of data that can be fed into a machine learning model, which may hinder the model's accuracy. While playing enough games using our model will eventually increase the accuracy, the quality of data is equally important, and balancing data quantity, quality, and project timeline is crucial.

A potential extension of this project would be to consider methods of quicker learning for our network. For example, if we can identify situations where either player has 3 chips in a row horizontally or vertically, then we can force the network to choose the winning move. Although our code may run slower, exposing the network to these scenarios will improve the accuracy over time. We can see that the majority of wins were obtained from vertical chips instead of horizontal or diagonal. This supports the idea that more board layouts should be explored by the model.

Conclusion

Our project focused on developing agents that excel at playing PopOut using strategic gameplay and decision-making via reinforcement learning (RL). We found RL to be a powerful technique for optimizing game strategies and demonstrated its potential effectiveness in developing game opponents. Our results showed that our RL agent achieved high performance in the challenging game of PopOut, indicating that investing in RL for game development can provide a competitive advantage in the gaming industry.

However, RL is a complex and computationally demanding field, and we faced challenges that highlighted the need for specialized expertise in RL to fully realize its benefits in our gaming programs. Additionally, obtaining satisfactory results can require substantial time and computational resources.

Despite the challenges, we believe that the potential of RL in game development outweighs the obstacles. Our project introduced us to the world of RL, particularly with policy gradients, and opened doors to innovative approaches in creating game opponents. We recommend investing in exploring and utilizing RL for game development to set us apart from other gaming companies and contribute to our success in the industry.