

Reinforcement Learning for Connect Four By Group 7

Introduction

Connect Four is a well-known two-player strategy game. Each player has their own colored discs and players alternate in selecting a slot on the board to drop the disc in with the goal of getting four of their colored discs in a row, either horizontal, vertical or diagonal. The game's roots stem from Tic-Tac-Toe, a pen and paper game with the goal of getting three in a row, as opposed to four.

Connect Four is a compelling game as although it appears simple and straightforward, there is significant opportunity to use strategy to increase and even guarantee one's likelihood of winning. For example, targeting specific slots over others is important. The slots in the middle of the board are more valuable than the ones on the edges because there is a higher chance of creating four in a row. The board and rules of the game enables a large variety of possible final boards with players able to take numerous different actions to reach them. We aimed to use reinforcement learning to find the optimal policies for the Connect Four

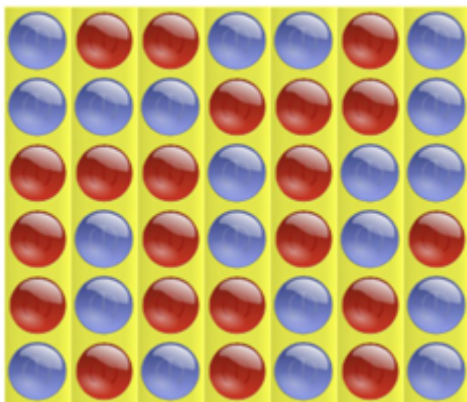
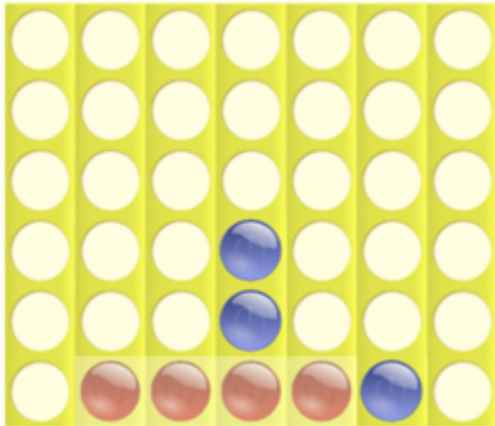
Connect Four Rules



Connect Four is played on a vertical board with six rows and seven columns which totals in 42 playable positions on the board. At the top vertical edge of the board, for each column there is a slit where the pieces are slotted into. Once the piece has been slotted into a column it falls down the column to the lowest row or it lands the row above the piece that was last played in that column. It is a two player game and each player has twenty one coin-like pieces in a different color to the other player.

The game begins by randomly deciding whether player one or player two will take the first turn. After the first player takes their turn, turns are then taken alternatively between the two players. A turn consists of a player dropping their colored piece into a column on the board. Each turn a player must drop their colored piece into a column, they cannot 'miss their turn.' If a column is full (six pieces in the column), then the piece cannot be played into that column. Additionally once the piece has been played, it cannot be undone or removed from the board.

The objective of Connect Four is to be the first player to connect four of their colored pieces in a row (either vertically, horizontally or diagonally) with no gaps on the board between the four pieces. At this point the game is over and the player who connected the four pieces wins. Alternatively, if all of the pieces have been played, resulting in the board being full, then the game is called a tie.



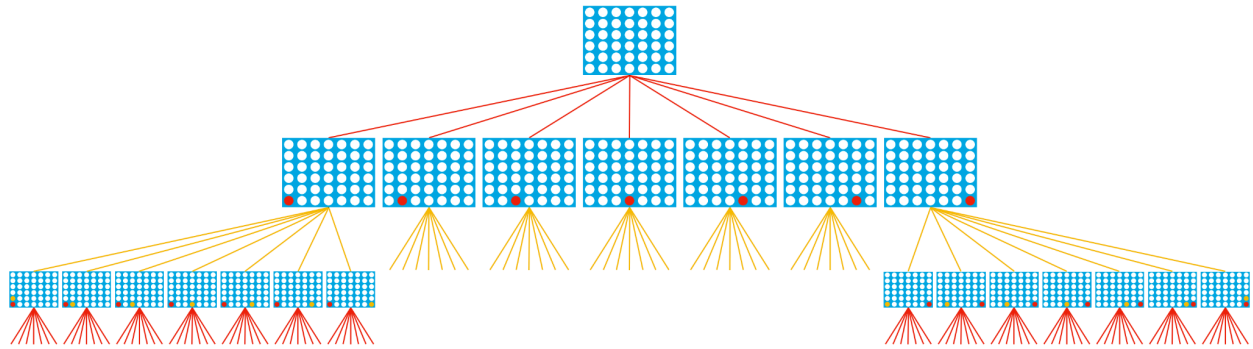
One-Step Lookahead

This step examines the one-step lookahead strategy, a decision-making tool used in game theory. The methodology involves simulating a single future move to determine the immediate value of different actions, facilitating optimal short-term decisions.

Game Trees

As a human player, we do a bit of forecasting before or while playing the game. For each potential move we predict what our opponent is likely to do in response, along with how we would then respond, and what the opponent is likely to do then, and so on. Then we choose the move that we think is most likely to result in a win.

We can formalize this idea and represent all possible outcomes in a complete Game Tree.



The game tree represents each possible move (by agent and opponent), starting with an empty board. The first row shows all possible moves the agent (red player) can make. Next, we record each move the opponent (yellow player) can make in response, and so on, until each branch reaches the end of the game. (The game tree for Connect Four gets quite large, so we show only a small preview in the image above.)

Once we can see every way the game can possibly end, it can help us to pick the move where we are most likely to win.

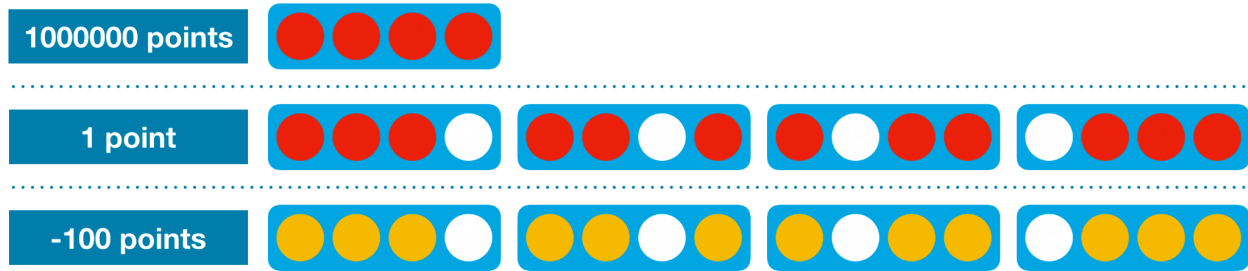
Heuristics

The complete game tree for Connect Four has over 4 trillion different boards! So in practice, our agent only works with a small subset when planning a move.

To make sure the incomplete tree is still useful to the agent, we will use a heuristic (or heuristic function). The heuristic assigns scores to different game boards, where we estimate that boards with higher scores are more likely to result in the agent winning the game. You will design the heuristic based on your knowledge of the game.

For instance, one heuristic that might work reasonably well for Connect Four looks at each group of four adjacent locations in a (horizontal, vertical, or diagonal) line and assigns:

- 1000000 (1e6) points if the agent has four discs in a row (the agent won),
- 1 point if the agent filled three spots, and the remaining spot is empty (the agent wins if it fills in the empty spot), and
- -100 points if the opponent fills three spots, and the remaining spot is empty (the opponent wins by filling in the empty spot).



How will the agent apply the heuristic? Imagine it's the agent's turn to make a move on the game board depicted at the top of the figure below. There are seven potential moves, corresponding to each column. For each possible move, the resulting game board configuration is recorded.

We then apply the heuristic to assign a score to each board by scanning the grid for all instances of the patterns specified in the heuristic, akin to solving a word search puzzle. Each found pattern affects the score accordingly. For example:

- The first board, where the agent places its piece in column 0, receives a score of 2. This score is derived from two separate patterns present on the board that each contribute one point, as highlighted in the image above.
- The second board is given a score of 1.
- The third board, where the agent plays in column 2, is scored at 0 because it lacks any of the heuristic's patterns.

With the highest score, the first board is the agent's chosen move, representing the best scenario for a win in the subsequent turn. Verify this by reviewing the figure now to ensure it aligns with your understanding.

This heuristic proves effective in this instance as it accurately identifies the optimal move. While this is just one of many possible heuristics for designing a Connect Four agent, experimenting with different heuristics might lead to even better results.

Code:

Our one-step lookahead agent will:

- Utilize the heuristic to evaluate and assign a score to each potential valid move.
- Choose the move that receives the highest score. If there are several moves tied for the highest score, one will be selected at random.

The term "one-step lookahead" describes the agent's strategy of considering only the immediate next move (one step ahead) rather than exploring further into the game tree.

We defined the agent using the functions given below.

```

# Calculates score if agent drops piece in selected column
def score_move(grid, col, mark, config):
    next_grid = drop_piece(grid, col, mark, config)
    score = get_heuristic(next_grid, mark, config)
    return score

# Helper function for score_move: gets board at next step if agent drops piece in selected column
def drop_piece(grid, col, mark, config):
    next_grid = grid.copy()
    for row in range(config.rows-1, -1, -1):
        if next_grid[row][col] == 0:
            break
    next_grid[row][col] = mark
    return next_grid

# Helper function for score_move: calculates value of heuristic for grid
def get_heuristic(grid, mark, config):
    num_threes = count_windows(grid, 3, mark, config)
    num_fours = count_windows(grid, 4, mark, config)
    num_threes_opp = count_windows(grid, 3, mark%2+1, config)
    score = num_threes - 1e2*num_threes_opp + 1e6*num_fours
    return score

```

The one-step lookahead agent is defined in the next code cell.

```

def agent(obs, config):
    # Get list of valid moves
    valid_moves = [c for c in range(config.columns) if obs.board[c] == 0]
    # Convert the board to a 2D grid
    grid = np.asarray(obs.board).reshape(config.rows, config.columns)
    # Use the heuristic to assign a score to each possible board in the next turn
    scores = dict(zip(valid_moves, [score_move(grid, col, obs.mark, config) for col in valid_moves]))
    # Get a list of columns (moves) that maximize the heuristic
    max_cols = [key for key in scores.keys() if scores[key] == max(scores.values())]
    # Select at random from the maximizing columns
    return random.choice(max_cols)

```

In the agent's code, we start by identifying the valid moves. Next, the game board is transformed into a 2D numpy array suitable for Connect Four, which consists of 6 rows and 7 columns.

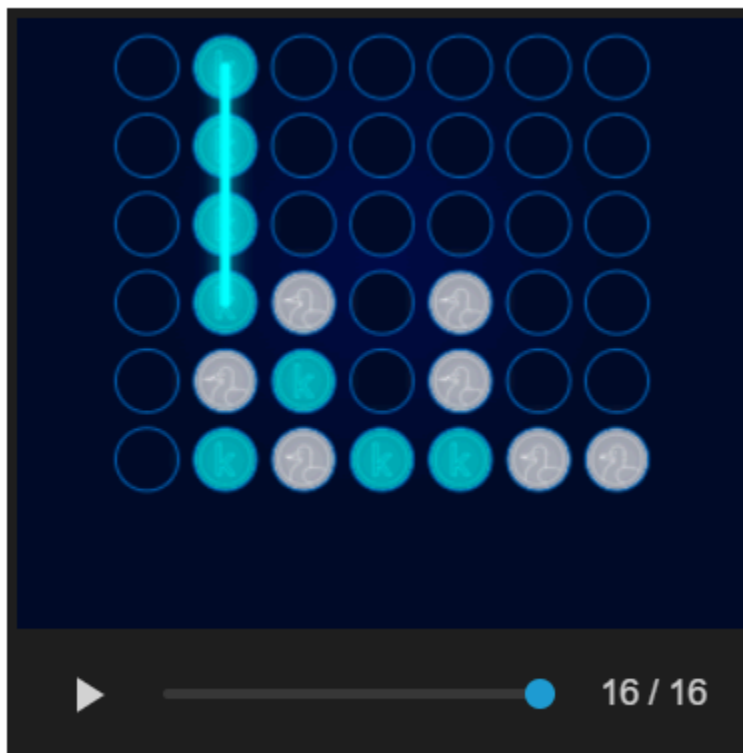
The function `score_move()` then evaluates the heuristic value for each valid move by utilizing several helper functions:

- `drop_piece()` generates a new grid depicting the scenario where the agent's disc is placed in a chosen column.
- `get_heuristic()` determines the heuristic value for the resulting board, where `mark` is the agent's symbol. This function calls `count_windows()`, which counts the number of

sequences (windows) of four adjacent slots (horizontally, vertically, or diagonally) that meet specific heuristic criteria. For example:

- Using `num_discs=4` and `piece=obs.mark` counts the occurrences of the agent aligning four discs in a row.
- Setting `num_discs=3` and `piece=obs.mark%2+1` counts the scenarios where the opponent has three discs, with the remaining slot empty—a potential setup for the opponent to win by occupying this space.

Finally, the agent identifies the columns that maximize the heuristic value and selects one at random to make its move.



We use the `get_win_percentage()` function to check how we can expect it to perform on average.

Agent 1 Win Percentage: 0.96

Agent 2 Win Percentage: 0.04

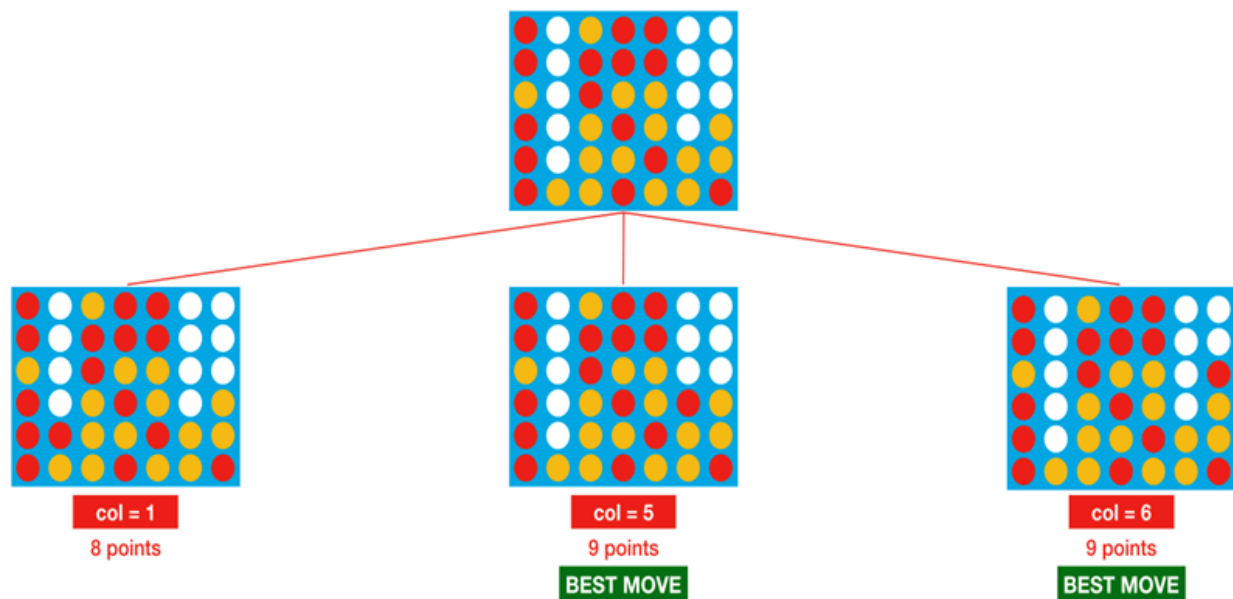
Number of Invalid Plays by Agent 1: 0

Number of Invalid Plays by Agent 2: 0

N-Step Lookahead

In the previous slides, you learned how to build an agent with one-step lookahead. This agent performs reasonably well, but definitely still has room for improvement!

For instance, consider the potential moves in the figure below. (Note that we have used zero-based numbering where the leftmost column corresponds to col=0 with the last column being col = 6.)

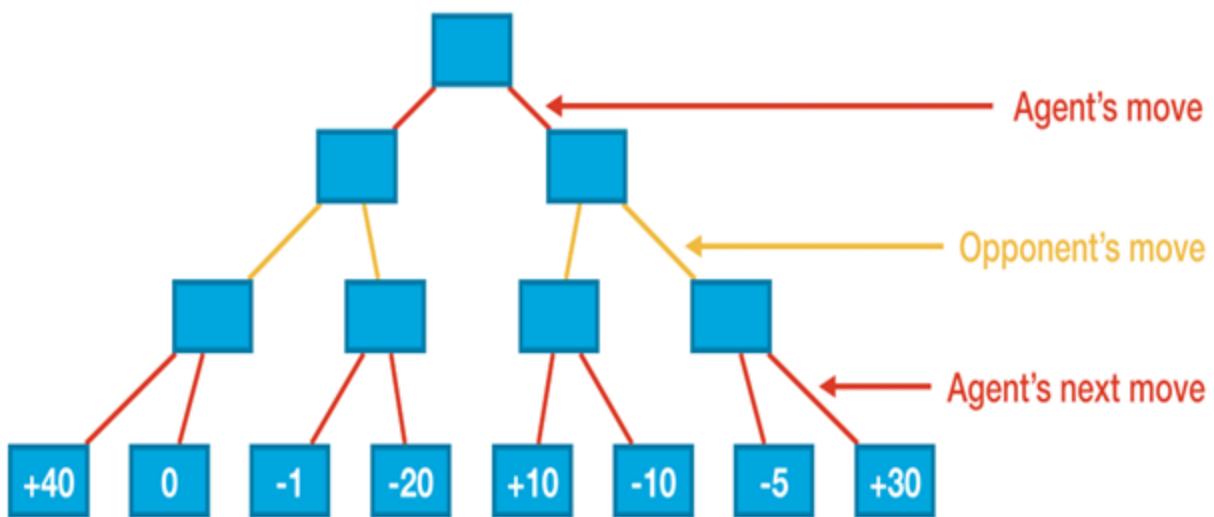


With one-step lookahead, the red player picks one of column 5 or 6, each with 50% probability as it has more reward associated. But column 5 is clearly a bad move, as it lets the opponent win the game in only one more turn. Unfortunately, the agent doesn't know this, because it can only look one move into the future.

Hence, what we thought of as the next step was to overcome this issue and use the minimax algorithm to help the agent look farther into the future and make better-informed decisions.

MiniMax Algorithm

Assume we're working with a depth of 3. This way, when choosing on a move, the agent examines all conceivable game boards that could result from the agent's move, the opponent's move, and the agent's subsequent move. For simplicity, we assume that at each turn, both the agent and opponent have only two possible moves. Each of the blue rectangles in the figure below corresponds to a different game board.



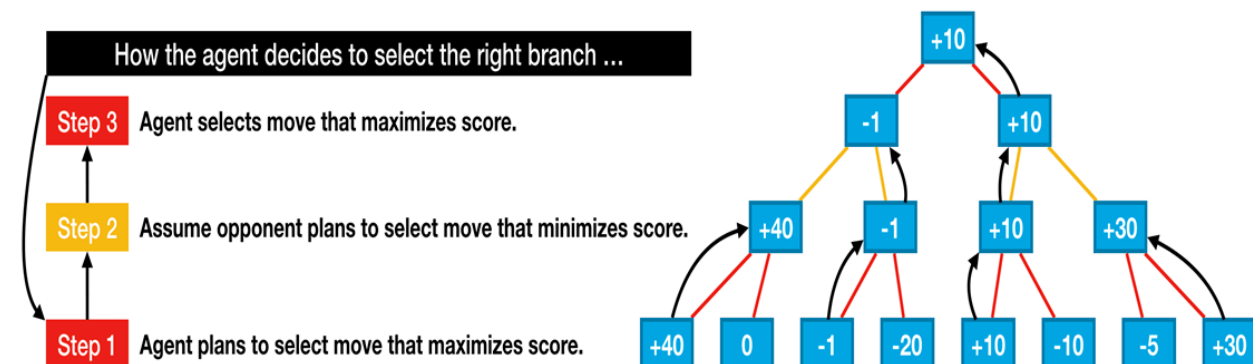
Labeled scores from the heuristic mentioned in previous slides with agent's goal being to end up with a score that's as high as possible.

But the agent has no complete control over its score and is dependent on the opponent's move to attain big scores and thus the opponent's selection can prove disastrous for the agent in some cases. For instance, If the agent picks the left branch, the opponent can force a score of -1.

You would see that the right branch is the better agent, since it is the less risky option as it gives up the score (+40) that can be accessed on the left, but it also guarantees at least +10 points on the right.

This is the main idea behind the **minimax algorithm**: where the agent chooses moves to get a high score possible assuming the opponent will counteract to move the score to be as low as possible.











Agent's Thinking process while selecting a move



Transition into Minimax Approach using Minimax Heuristics

We'll need to slightly modify the heuristic from the previous slides, since the opponent is now able to modify the game board results based upon its move.

We need to check if the opponent has won the game by playing a disc. The new heuristic looks at each group of four adjacent locations in a (horizontal, vertical, or diagonal) line and assigns:

1000000 points	
1 point	   
-100 points	   
-10000 points	

- **1000000 (1e6) points** if the agent has four discs in a row (the agent won),
- **1 point** if the agent filled three spots, and the remaining spot is empty (the agent wins if it fills in the empty spot),
- **-100 points** if the opponent filled three spots, and the remaining spot is empty (the opponent wins by filling in the empty spot), and
- **-10000 (-1e4) points** if the opponent has four discs in a row (the opponent won).

Coding implementation for Minimax algorithm

Helper function for minimax: calculates value of heuristic for grid

```
def get_heuristic(grid, mark, config):  
    num_threes = count_windows(grid, 3, mark, config)  
    num_fours = count_windows(grid, 4, mark, config)  
    num_threes_opp = count_windows(grid, 3, mark%2+1, config)  
    num_fours_opp = count_windows(grid, 4, mark%2+1, config)  
    score = num_threes - 1e2*num_threes_opp - 1e4*num_fours_opp +  
    1e6*num_fours  
    return score
```

In this function above, we would calculate the heuristic scores by counting the number of three pairs, four pairs we have for our agent and the opponent and would calculate the score based upon that for the entire grid.

In the next code cell, we define a few additional functions that we'll need for the minimax agent.

```
# Uses minimax to calculate value of dropping piece in selected column
```

```
def score_move(grid, col, mark, config, nsteps):  
    next_grid = drop_piece(grid, col, mark, config)  
    score = minimax(next_grid, nsteps-1, False, mark, config)  
    return score
```

This function above calculates the score of dropping a piece in a selected column using a heuristic evaluation. The heuristic evaluation is a way to assess the "goodness" of a move based on the current state of the board.

```
# Helper function for minimax: checks if agent or opponent has four in a row in the window
```

```
def is_terminal_window(window, config):  
    return window.count(1) == config.inarow or window.count(2) == config.inarow
```

This function above checks if a window (a sequence of adjacent cells) contains four pieces in a row belonging to either the player or the opponent.

```
# Helper function for minimax: checks if game has ended
```

```
def is_terminal_node(grid, config):  
    # Check for draw  
    if list(grid[0, :]).count(0) == 0:  
        return True  
  
    # Check for win: horizontal, vertical, or diagonal  
    # horizontal  
    for row in range(config.rows):  
        for col in range(config.columns-(config.inarow-1)):  
            window = list(grid[row, col:col+config.inarow])  
            if is_terminal_window(window, config):  
                return True  
  
    # vertical  
    for row in range(config.rows-(config.inarow-1)):
```

```

        for col in range(config.columns):
            window = list(grid[row:row+config.inarow, col])
            if is_terminal_window(window, config):
                return True

    # positive diagonal
    for row in range(config.rows-(config.inarow-1)):
        for col in range(config.columns-(config.inarow-1)):
            window = list(grid[range(row, row+config.inarow), range(col,
col+config.inarow)])
            if is_terminal_window(window, config):
                return True

    # negative diagonal
    for row in range(config.inarow-1, config.rows):
        for col in range(config.columns-(config.inarow-1)):
            window = list(grid[range(row, row-config.inarow, -1),
range(col, col+config.inarow)])
            if is_terminal_window(window, config):
                return True

    return False

```

This function above checks if the game has ended by either a win, loss, or draw.

Minimax implementation

```

def minimax(node, depth, maximizingPlayer, mark, config):
    is_terminal = is_terminal_node(node, config)
    valid_moves = [c for c in range(config.columns) if node[0][c] == 0]
    if depth == 0 or is_terminal:
        return get_heuristic(node, mark, config)

```

```

    if maximizingPlayer:
        value = -np.Inf
        for col in valid_moves:
            child = drop_piece(node, col, mark, config)
            value = max(value, minimax(child, depth-1, False, mark,
config))
        return value
    else:
        value = np.Inf
        for col in valid_moves:
            child = drop_piece(node, col, mark%2+1, config)
            value = min(value, minimax(child, depth-1, True, mark,
config))
        return value

```

This function above implements the minimax algorithm, which is a recursive algorithm used to choose the best move for the current player, assuming the opponent also plays optimally.

Explanation of the function

If the depth is 0 or the game has ended, it returns the heuristic value of the node.

After that we would have two cases where **firstly, it is thinking about maximizing its output when it is his turn and secondly, it is minimizing the output when it is opponents turn**

If it's the maximizing player's turn, it calculates the maximum possible score for the current node by recursively calling minimax for all possible moves.

If it's the minimizing player's turn, it calculates the minimum possible score for the current node by recursively calling minimax for all possible opponent moves.

Testing MiniMax with the random agent

```
# How deep to make the game tree: higher values take longer to run!
```

```
N_STEPS = 3
```

```
def agent(obs, config):
```

```
    # Get list of valid moves
```

```

    valid_moves = [c for c in range(config.columns) if obs.board[c] ==
0]

    # Convert the board to a 2D grid
    grid = np.asarray(obs.board).reshape(config.rows, config.columns)

    # Use the heuristic to assign a score to each possible board in the
next step
    scores = dict(zip(valid_moves, [score_move(grid, col, obs.mark,
config, N_STEPS) for col in valid_moves]))

    # Get a list of columns (moves) that maximize the heuristic
    max_cols = [key for key in scores.keys() if scores[key] ==
max(scores.values())]

    # Select at random from the maximizing columns
    return random.choice(max_cols)

```

The function above selects a move based on a heuristic evaluation of the current board state. It assesses each possible move's desirability, chooses the moves that maximize the heuristic score, and randomly selects one of them.

Testing MiniMax with the random agent

```

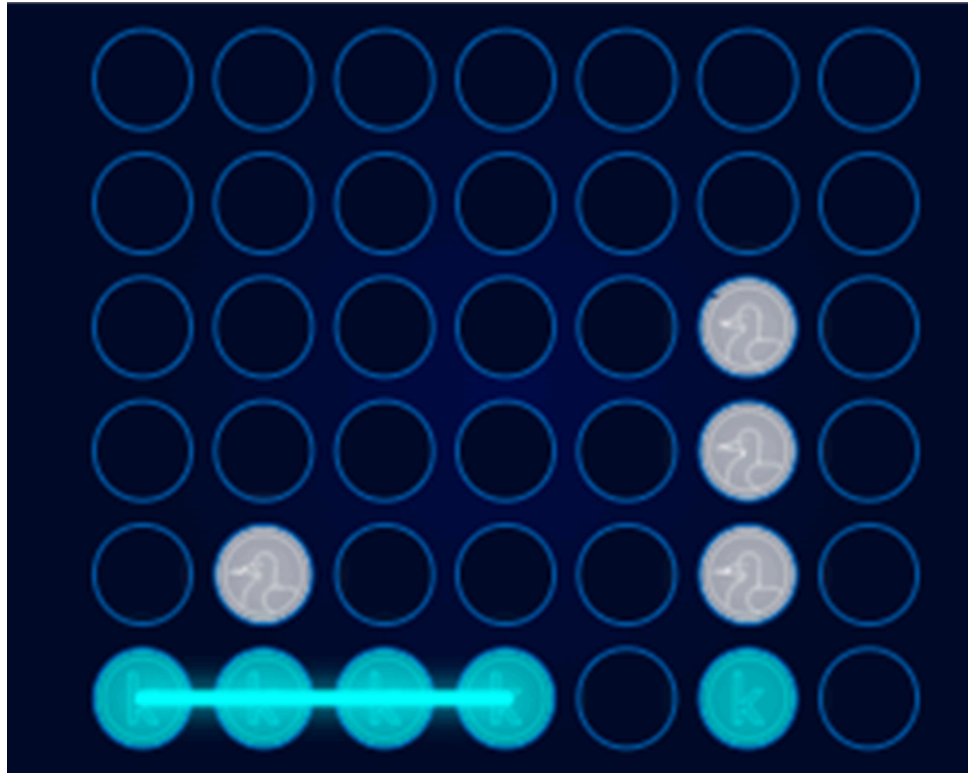
from kaggle_environments import make, evaluate

# Create the game environment
env = make("connectx")

# Two random agents play one game round
env.run([agent, "random"])

# Show the game
env.render(mode="ipython")

```



Check the model accuracy with win percentage

```
get_win_percentages(agent1=agent, agent2="random", n_rounds=50)
```

Agent 1 Win Percentage: 1.0

Agent 2 Win Percentage: 0.0

Number of Invalid Plays by Agent 1: 0

Number of Invalid Plays by Agent 2: 0

Deep Reinforcement Learning

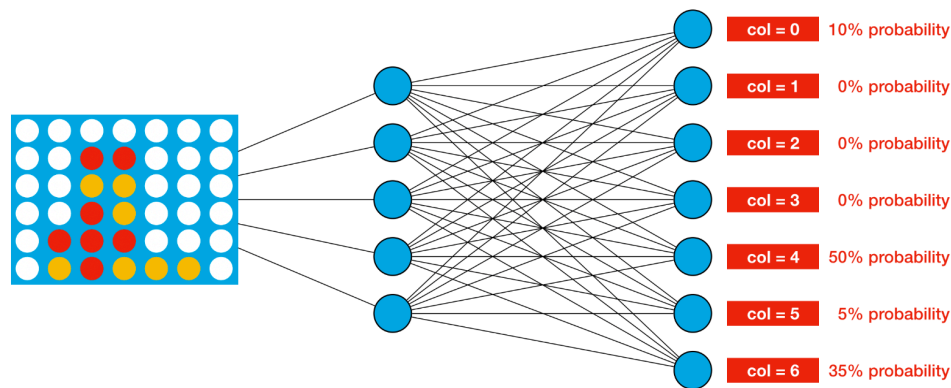
Up to this point, agents have been dependent on precise instructions on how to play the game, with the heuristics playing a crucial role in guiding move selection.

Neural Networks

Creating an ideal heuristic is a challenging task. To refine it, we often need to play numerous games to pinpoint situations where the agent might have opted for better moves. Moreover, diagnosing the precise issues and correcting past errors without inadvertently creating new ones can be quite difficult.

Wouldn't a more structured method for enhancing the agent based on gameplay experience be simpler?

To achieve this we substitute the heuristic with a neural network. This network will take the current board as input and output a probability for each potential move, offering a systematic approach to improving the agent's decision-making.



Next, the agent chooses a move by sampling from these probabilities. For example, with the game board shown in the image above, the agent might select column 4 with a 50% probability.

The strategy here is to simply adjust the network's weights so that it consistently assigns higher probabilities to the more advantageous moves for every possible game board configuration. That's the theory, at least. In practice, it is difficult to verify this directly especially considering that Connect Four has over 4 trillion possible board configurations!

Setup

Here is the method we used to modify the weights of the network. After each move, the agent receives a reward indicating its performance:

- If the agent wins the game with that move, it receives a reward of +1.
- If the agent makes an invalid move (ending the game), it receives a reward of -10.
- If the opponent wins on their next move (meaning the agent failed to block a winning move), the agent receives a reward of -1.
- For any other outcome, the agent receives a reward of 1/42.

At the end of each game, the agent tallies its rewards, known as its cumulative reward. For example:

- If a game lasts 8 moves (with each player moving four times) and the agent wins, its cumulative reward would be $3 \cdot (1/42) + 1$.
- If a game goes 11 moves (with the opponent starting, and the agent playing five times), and the opponent wins on their last move, the agent's cumulative reward would be $4 \cdot (1/42) - 1$.
- If the game ends in a draw after 21 moves by the agent, the cumulative reward is $21 \cdot (1/42)$.
- If the game ends in 7 moves due to the agent making an invalid move, the cumulative reward is $3 \cdot (1/42) - 10$.

Our objective is to adjust the neural network's weights to maximize the agent's cumulative reward.

This approach of using rewards to gauge an agent's performance is fundamental in reinforcement learning. By framing the problem this way, we can apply various reinforcement learning algorithms to develop our agent.

Reinforcement Learning:

There are various reinforcement learning algorithms available, such as DQN, A2C, and PPO. These methods generally follow a common framework to develop an agent:

1. The initial weights of the network are set randomly.
2. As the agent engages in gameplay, the algorithm experiments with different weight configurations to observe their impact on the agent's cumulative reward. Over numerous games, this process helps clarify how the weights influence the reward, leading the algorithm to favor weights that yield better performance.

While this overview simplifies the complexities involved, our focus here is on the broader concept. This approach aims to produce an agent that maximizes its chances of winning (thereby earning the final reward of +1, while avoiding penalties of -1 and -10) and seeks to extend the duration of the game (to accumulate as many $1/42$ bonuses as possible).

It might seem counterintuitive to design a game strategy that prolongs gameplay, possibly delaying obvious winning moves. This might lead to an agent that appears inefficient. The inclusion of the $1/42$ bonus is not primarily about extending game length but is a strategy to aid algorithm convergence.

Proximal Policy Optimization (PPO)

We employed the Proximal Policy Optimization (PPO) algorithm to develop our agent. PPO is a type of policy gradient method for reinforcement learning which balances the benefits of on-policy learning with the efficiency of off-policy methods. It works by updating policies in a way that doesn't diverge too drastically from the previous policy, using a clipped probability ratio, which prevents large updates. This ensures stable and reliable improvement, making PPO popular for its robust performance and simplicity in tuning. In our application, PPO will help the

agent refine its decision-making strategy incrementally to maximize its cumulative reward efficiently.

Code:

We will be using Stable-Baselines3 for implementation of reinforcement learning algorithm. Stable Baselines3 is a set of reliable implementations of reinforcement learning (RL) algorithms in Python, built on top of the popular PyTorch framework.

To integrate Stable Baselines with our setup, we make some adjustments to the environment. For this purpose we have created the `ConnectFourGym` class. This class adapts ConnectX to function as an OpenAI Gym environment and includes several methods:

- `reset()` will be invoked at the start of each game, returning the initial game board as a 2D numpy array with 6 rows and 7 columns.
- `change_reward()` alters the rewards the agent receives to align with the reward system we have devised. This is separate from the competition's standard reward mechanism, which ranks agents.
- `step()` takes the agent's selected action (provided as action) and the opponent's response, and returns:
 - The updated game board (as a numpy array),
 - The reward for the agent based on its most recent move (either +1, -10, -1, or 1/42),
 - A status indicator of whether the game has concluded (`done=True` if the game is over; otherwise, `done=False`).

```

class ConnectFourGym(gym.Env):
    def __init__(self, agent2="random"):
        ks_env = make("connectx", debug=True)
        self.env = ks_env.train([None, agent2])
        self.rows = ks_env.configuration.rows
        self.columns = ks_env.configuration.columns
        # Learn about spaces here: http://gym.openai.com/docs/#spaces
        self.action_space = spaces.Discrete(self.columns)
        self.observation_space = spaces.Box(low=0, high=2,
                                            shape=(1,self.rows,self.columns), dtype=int)

        # Tuple corresponding to the min and max possible rewards
        self.reward_range = (-10, 1)
        # StableBaselines throws error if these are not defined
        self.spec = None
        self.metadata = None

    def reset(self):
        self.obs = self.env.reset()
        return np.array(self.obs['board']).reshape(1,self.rows,self.columns)

    def change_reward(self, old_reward, done):
        if old_reward == 1: # The agent won the game
            return 1
        elif done: # The opponent won the game
            return -1
        else: # Reward 1/42
            return 1/(self.rows*self.columns)

    def step(self, action):
        # Check if agent's move is valid
        is_valid = (self.obs['board'][int(action)] == 0)
        if is_valid: # Play the move
            self.obs, old_reward, done, _ = self.env.step(int(action))
            reward = self.change_reward(old_reward, done)
        else: # End the game and penalize agent
            reward, done, _ = -10, True, {}
        return np.array(self.obs['board']).reshape(1,self.rows,self.columns), reward, done, _

```

The next step involves defining the neural network's architecture. For our purposes, we are employing a convolutional neural network (CNN).

This CNN is responsible for outputting the probabilities associated with selecting each column. As we are utilizing the Proximal Policy Optimization (PPO) algorithm (highlighted in the code cell below), our network also produces additional output, specifically the "value" of the input. This concept is part of the broader framework of "actor-critic networks".

Why use a Convolutional Neural Network (CNN)?

In the context of reinforcement learning for grid-based games like Connect Four, a Convolutional Neural Network is particularly effective due to several reasons:

- **Spatial Hierarchy:** CNNs are adept at handling data with a spatial hierarchy, where local patterns form the basis for more complex structures. In games like Connect Four, understanding local relationships—such as the arrangement of discs—can be crucial for assessing the game state.

- **Feature Extraction:** CNNs automatically detect and use relevant features without needing manual specification. They can learn to recognize patterns such as potential four-in-a-rows, threats, and opportunities, which are essential for making strategic decisions in grid-based games.
- **Efficiency:** By using convolutional layers, CNNs reduce the number of parameters needed, making them more computationally efficient compared to fully connected networks, especially when dealing with larger grids
- **Adaptability:** The same architecture can generally be adapted for similar types of games (like tic-tac-toe or chess), making CNNs versatile tools in the game-playing field of AI.

Overall, the choice of a CNN allows the model to interpret the game board as a two-dimensional image, effectively recognizing and evaluating patterns that are critical for strategic gameplay in Connect Four. This ability to process and learn from visual input makes CNNs ideal for this application in reinforcement learning.

```

# Neural network for predicting action values
class CustomCNN(BaseFeaturesExtractor):

    def __init__(self, observation_space: gym.spaces.Box, features_dim: int=128):
        super(CustomCNN, self).__init__(observation_space, features_dim)
        # CxHxW images (channels first)
        n_input_channels = observation_space.shape[0]
        self.cnn = nn.Sequential(
            nn.Conv2d(n_input_channels, 32, kernel_size=3, stride=1, padding=0),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=0),
            nn.ReLU(),
            nn.Flatten(),
        )

        # Compute shape by doing one forward pass
        with th.no_grad():
            n_flatten = self.cnn(
                th.as_tensor(observation_space.sample()[None]).float()
            ).shape[1]

        self.linear = nn.Sequential(nn.Linear(n_flatten, features_dim), nn.ReLU())

    def forward(self, observations: th.Tensor) -> th.Tensor:
        return self.linear(self.cnn(observations))

policy_kwargs = dict(
    features_extractor_class=CustomCNN,
)

# Initialize agent
model = PPO("CnnPolicy", env, policy_kwargs=policy_kwargs, verbose=0)

```

In the above, the weights of the neural network are initially set to random values.

In the next code cell, we train the agent which is just another way of saying that we find weights of the neural network that are likely to result in the agent selecting good moves.

```

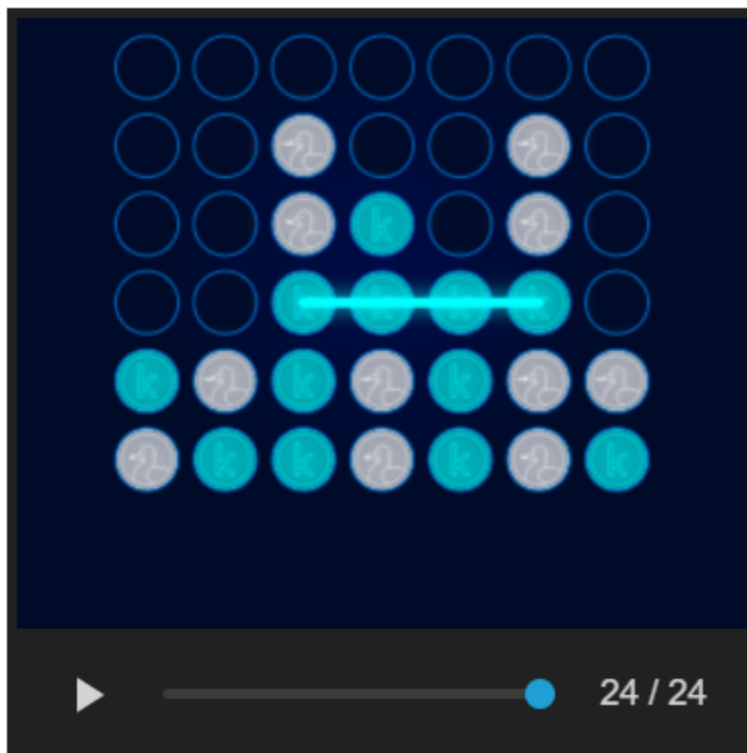
# Train agent
model.learn(total_timesteps=60000)

```

Specify the agent trained

```
def agent1(obs, config):
    # Use the best model to select a column
    col, _ = model.predict(np.array(obs['board']).reshape(1, 6,7))
    # Check if selected column is valid
    is_valid = (obs['board'][int(col)] == 0)
    # If not valid, select random move.
    if is_valid:
        return int(col)
    else:
        return random.choice([col for col in range(config.columns) if obs.board[int(col)] == 0])
```

The outcome of one game round against a random agent:



And we calculated how it performs on average against the random agent.

Agent 1 Win Percentage: 0.68

Agent 2 Win Percentage: 0.32

Number of Invalid Plays by Agent 1: 0

Number of Invalid Plays by Agent 2: 0

Above we show the performance statistics from a comparison between two game-playing agents in a simulation, where Agent 1 is powered by a neural network and Agent 2 operates on random decision-making. Here is a breakdown of the information:

- Agent 1 Win Percentage: 0.68 (0.68%) - This indicates that the neural network-based Agent 1 won 68% of the games played against Agent 2.
- Agent 2 Win Percentage: 0.32 (0.32%) - Agent 2, which makes decisions randomly, won 32% of the games.
- Number of Invalid Plays by Agent 1: 0 - Agent 1 did not make any invalid moves throughout the simulation, suggesting that the neural network effectively learned or was programmed to always choose valid moves.
- Number of Invalid Plays by Agent 2: 0 - Despite its random strategy, Agent 2 also did not make any invalid moves, which could be attributed to either a constraint in the testing environment or pure chance.

Overall, this demonstrates that the neural network (Agent 1) performs significantly better than the random Agent (Agent 2) in this context, achieving over twice the win rate. However, both agents showed a good understanding of the game rules, as indicated by their lack of invalid plays.