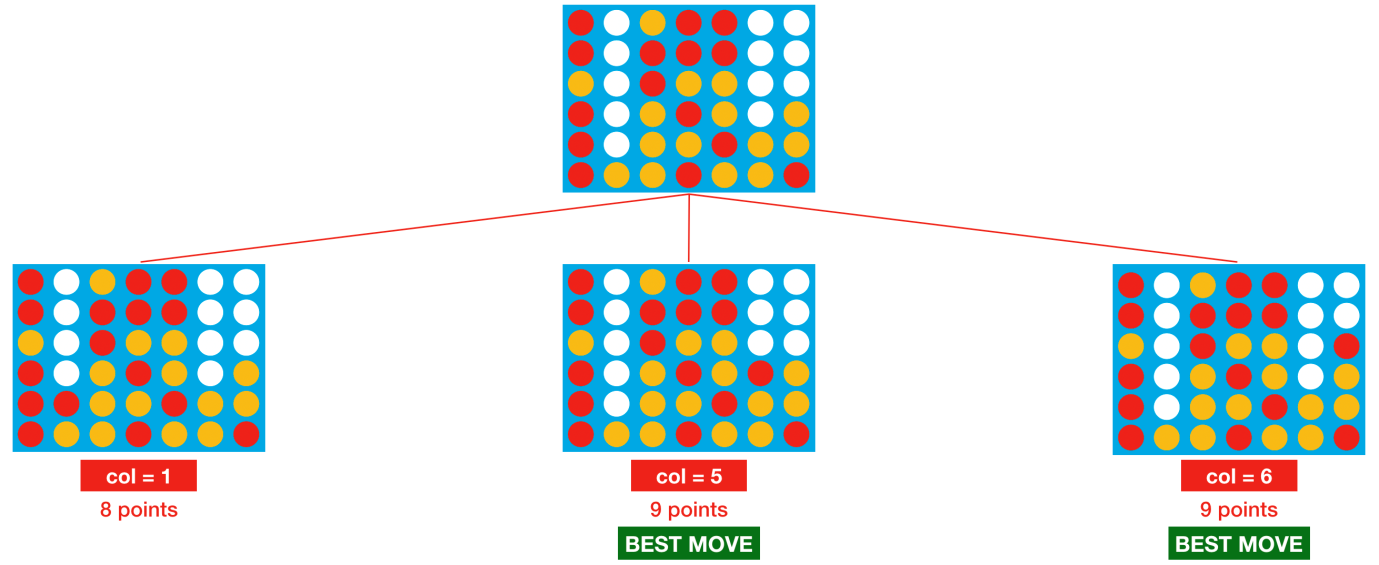
Gaming AI & Reinforcement Learning: 3rd lesson – N-step Lookahead

In the previous tutorial, 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 use zero-based numbering for the columns, so the leftmost column corresponds to col=0, the next column corresponds to col=1, and so on).



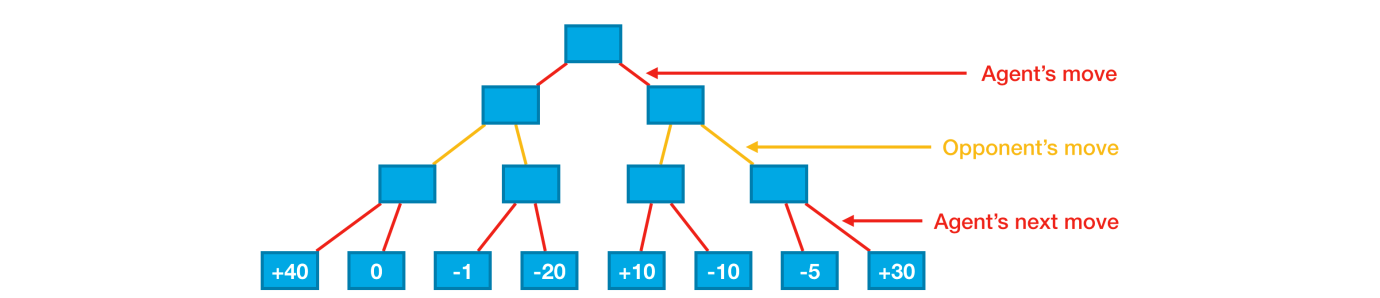
With one-step lookahead, the red player picks of column 5 or 6, each with 50% probability. But, column 5 is clearly a bad move, as it lets the opponent win the game in only one more turn. Unfortunately, the agent doesn’t know this, because it can only look one move into the future.

In this tutorial, you’ll use the **minimax algorithm** to help the agent look farther into the future and make better-informed decisions.

***Minimax***

We'd like to leverage information from deeper in the game tree. For now, assume we work with a depth of 3. This way, when deciding its move, the agent considers all possible game boards that can result from the agent’s move, the opponent’s move, and the agent’s next move.

We'll work with a visual example. 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.



We have labeled each of the ”leaf nodes” at the bottom of the tree with the score from the heuristic (we use made-up scores in the figure, in the code we'll use the same heuristic from the previous tutorial). As before, the current game board is at the top of the figure, and the agent's goal is to end up with a score that's as high as possible.

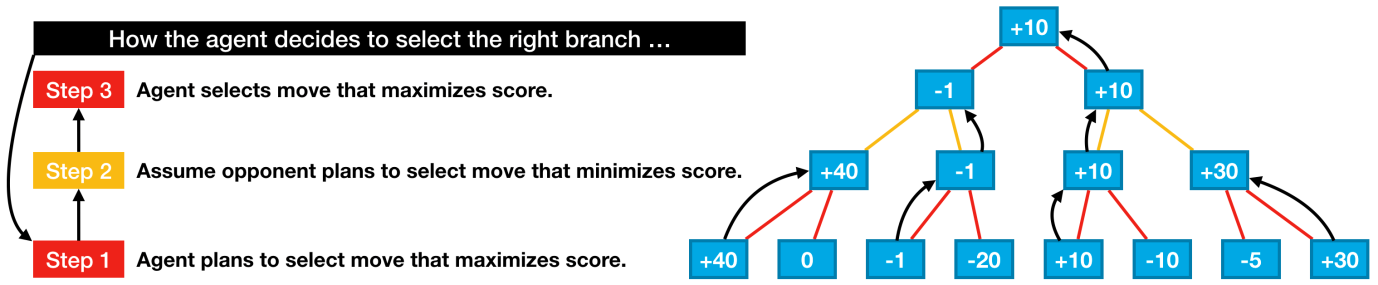
But notice that, the agent no longer has complete control over its score; after the agent makes its move, the opponent selects its own move. And, the opponent’s selection can prove disastrous for the agent! In particular:

* If the agent chooses the left branch, the opponent can force a score of -1.
* If the agent chooses the right branch, the opponent can force a score of 10.

Take the time now to check this in the figure, to make sure it makes sense to you! With this in mind, you might argue that the right branch is the better choice for the agent, since it is the less risky option. Sure, it gives up the possibility of getting the large score (40 points) that can be only accessed on the left branch, but it also guarantees that the agent gets at least 10 points.

This is the main idea behind the minimax algorithm; the agent chooses moves to get a score that is as high as possible, and it assumes the opponent will counteract this by choosing moves to force the score to be as low as possible. That is, the agent and opponent have opposing goals, and we assume the opponent plays optimally.

So, in practice, how does the agent use this assumption to select a move? We illustrate the agent's thought process in the figure below.



In the example, minimax assigns the move on the left a score of -1, and the move on the right is assigned a score of 10. So, the agent will select the move on the right.

***Code***

We'll use several functions from the previous tutorial. These are defined in the hidden code cell below.

import random

import numpy as np

*# 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 get\_heuristic: checks if window satisfies heuristic conditions*

def check\_window(window, num\_discs, piece, config):

return (window.count(piece) == num\_discs **and** window.count(0) == config.inarow-num\_discs)

*# Helper function for get\_heuristic: counts number of windows satisfying specified heuristic conditions*

def count\_windows(grid, num\_discs, piece, config):

num\_windows = 0

*# 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 check\_window(window, num\_discs, piece, config):

num\_windows += 1

*# 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 check\_window(window, num\_discs, piece, config):

num\_windows += 1

*# 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 check\_window(window, num\_discs, piece, config):

num\_windows += 1

*# negative diagonal*

for row **in** range(config.inarow-1, config.rows):

for col **in** range(config.columns-(config.inarow-1)):

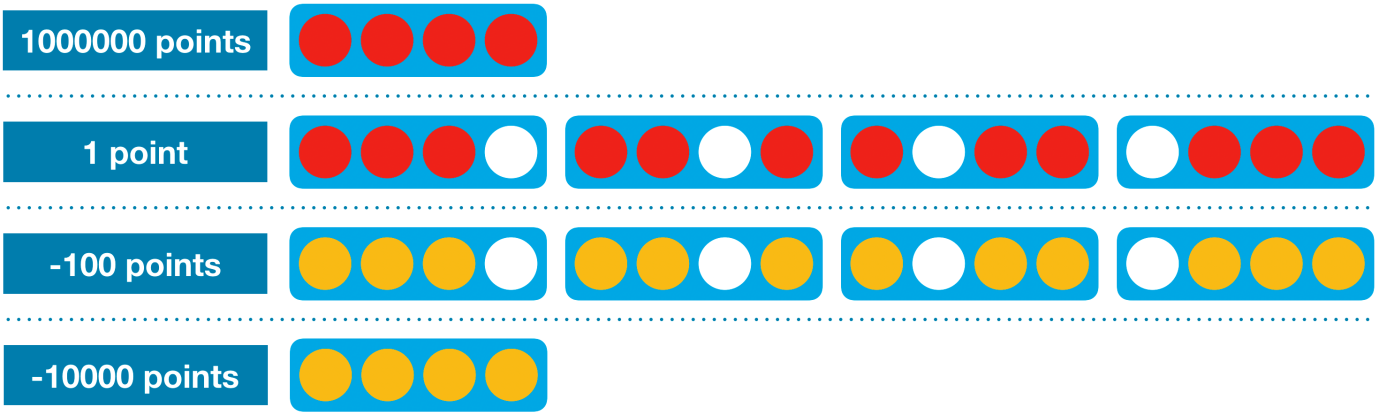
window = list(grid[range(row, row-config.inarow, -1), range(col, col+config.inarow)])

if check\_window(window, num\_discs, piece, config):

num\_windows += 1

return num\_windows

We'll also need to slightly modify the heuristic from the previous tutorial, since the opponent is now able to modify the game board.



In particular, 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:

* 1,000,000 (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 (-1e2) points if the opponent filled three spots, and the remaining spot is empty (the opponent wins by filling in the empty spot)
* -10,000 (-1e4) points if the opponent has four discs in a row (the opponent won)

This is defined in the code cell below.

*# 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 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

*# 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

*# 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

*# 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

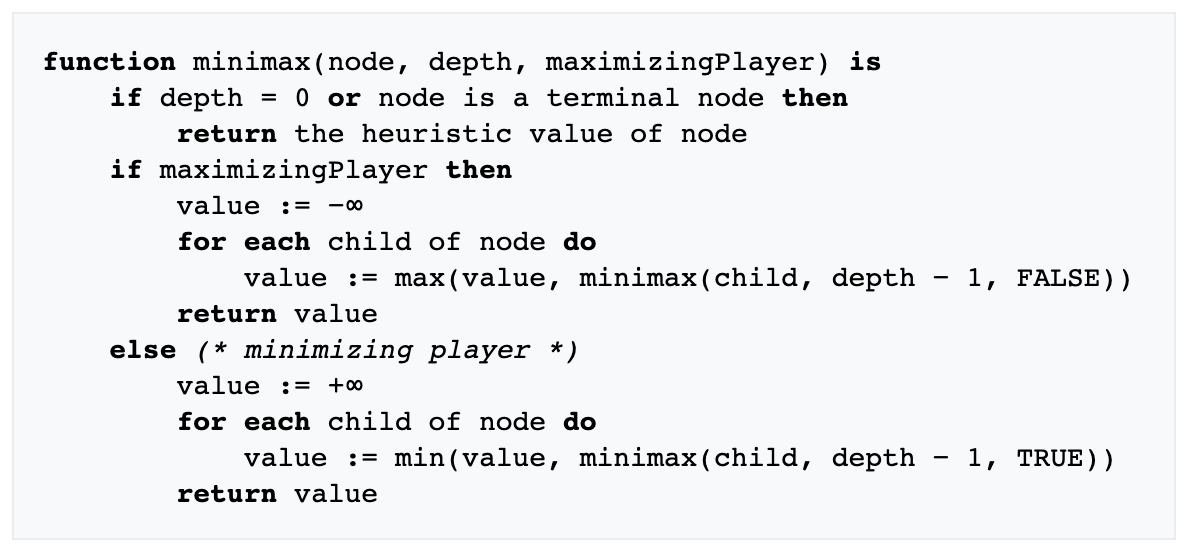
for col **in** valid\_moves:

child = drop\_piece(node, col, mark%2+1, config)

value = min(value, minimax(child, depth-1, True, mark, config))

return value

We won't describe the minimax implementation in detail, but if you want to read more technical pseudocode, here's the description from Wikipedia (note that the pseudocode can be safely skipped!).



Finally, we implement the minimax agent in the competition format. The N\_STEPS variable is used to set the depth of the tree.

*# 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)

In the next code cell, we see the outcome of one game round against a random agent.

from kaggle\_environments import make, evaluate

*# Create the game environment*

env = make("connectx")

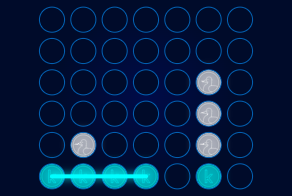
*# Two random agents play one game round*

env.run([agent, "random"])

*# Show the game*

env.render(mode="ipython")

Loading environment lux\_ai\_s2 failed: No module named 'vec\_noise'



And we check how we can expect it to perform on average. Not bad, isn’t it?

def get\_win\_percentages(agent1, agent2, n\_rounds=100):

*# Use default Connect Four setup*

config = {'rows': 6, 'columns': 7, 'inarow': 4}

*# Agent 1 goes first (roughly) half the time*

outcomes = evaluate("connectx", [agent1, agent2], config, [], n\_rounds//2)

*# Agent 2 goes first (roughly) half the time*

outcomes += [[b,a] for [a,b] **in** evaluate("connectx", [agent2, agent1], config, [], n\_rounds-n\_rounds//2)]

print("Agent 1 Win Percentage:", np.round(outcomes.count([1,-1])/len(outcomes), 2))

print("Agent 2 Win Percentage:", np.round(outcomes.count([-1,1])/len(outcomes), 2))

print("Number of Invalid Plays by Agent 1:", outcomes.count([None, 0]))

print("Number of Invalid Plays by Agent 2:", outcomes.count([0, None]))

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