Deep Reinforcement Learning for the Connect Four gaming application

University of California, Berkeley

Johan Gerfaux
IEOR Department

Stefano Zavagli
IEOR Department

***Team separation of work*: During the first phase of the project, Stefano modeled the mathematical structure of the game while Johan implemented the code of the optimization program. Once the foundations for these two parts were at a high degree of completion, we transitioned into closely collaboration in order to fine-tune and add as much value to the project as possible. In terms of the final report, both of us contributed equally to finalize it.**

I. Motivation

AlphaGo is one of the most notorious breakthroughs in terms of machine performance in board gaming applications. Silver's AlphaGo [6] provided strong evidence that certain algorithms are able to develop playing strategies comparable, and even superior to the best human experts. Deep Mind's feat shines a bright light on the relevance of Reinforcement Learning concepts for complex strategic tasks.

For our project, we attempt to apply Deep Reinforcement Learning techniques in the "Connect Four" board game. We utilize the platform and resources provided by a public Kaggle competition whereby users create their own agents, and those agents play the game against each other [3]. The competition provides certain templates for basic programming objects. We proceeded to set up the working environment according to the requirements of the competition and implemented our learning programs to produce agents as competitive as possible.

We then used the public platform to evaluate the effectiveness of our agents; that is, measure the performance achieved with our learning methods against agents created by other developers. The main goal of this exercise is to compare agents' performance and speed using different algorithms. Ideally we would like to see an incremental level of performance progression, culminating with the implementation of a Reinforcement Learning algorithm that achieves superior results.

II. The Game

Connect Four is a two-player board game, where players select colored checkers and drop them into positions with the aim of forming connections between the checkers. The objective for both players is to arrange four of their colored checkers either vertically, horizontally, or diagonally before their opponent does so. Each player takes one turn and drops a checker into one of the available columns. The playing grid consists of seven columns and six rows, i.e. 42 different board positions, and pieces fall down each column as they are dropped occupying the lowest available space within the selected column.

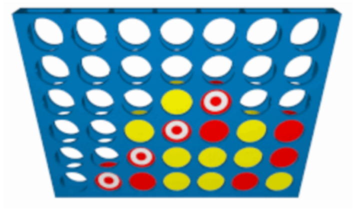


Figure 1: In-game example - red agent connecting four chips and producing a winning combination [3]

III. Mathematical Structure of the Game

A. The Optimization Problem

We wish to model the game as an optimization program. We start by describing the foundations of the model, and then proceed to describe some of the techniques that were implemented to solve the program. One key aspect worth noting from the get-go is that we want to create a quick, responsive agent, but also one with good decision-making capabilities. The Connect Four game presents 2^{42} combinations, i.e. ~4 trillion different possible board arrangements. Hence, we choose approaches that will attempt to mitigate the so-called "curse of dimensionality", while at the same time provide realistic and competitive gameplay moving patterns.

Approximate Dynamic Programming can help us achieve the balance between performance and time efficiency that we are looking for. Specifically, we pursued implementation of approximations in value space. Starting from the standard DP algorithm:

(1)

$$J_N(x_n) = g_N(x_N)$$

$$J_k(x_k) = \min_{u_k \in U_k(x_k)} \left\{ \mathbb{E}_{w_k} \left[g_k(x_k, u_k, w_k) + J_{k+1}(f_k(x_k, u_k, w_k)) \right] \right\}, \forall k \in \{0, ..., N-1\}$$

By incorporating an approximate value function, denoted as $\tilde{J}_{k+1}$, we substitute our exact value function J_{k+1} (also called the exact "cost-to-go").

Furthermore, we can represent explicitly the approximate value function of choice that will help us solve the optimization problem using heuristics such as Multistep Lookahead, and Deterministic Tree Search. The Multistep Lookahead equation is:

(3)

$$\tilde{J}_{k+1}(x_{k+1}) = \min_{(\mu_{k+1},\ldots,\mu_{k+l-1})} \mathbb{E}_{w_{k+1},\ldots,w_{k+l-1}} \left[\tilde{J}_{k+l}(x_{k+l}) + \sum_{i=k+1}^{k+l-1} g_i(x_i, \mu_i(x_i), w_i) \right]$$

Ultimately, we are able to obtain a sub-optimal admissible policy:

(4)

$$\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \left\{ \mathbb{E}_{w_k} \left[g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}(f_k(x_k, u_k, w_k)) \right] \right\}, \forall k \in \{0, \ldots, N-1\}$$

B. *Stage Cost*

Now we wish to define a cost function $g_k(x_k, \mu_k(x_k), w_k)$, and identify the possible disturbances that affect our system.

Since there are two agents playing the game, our model will be focused on the choices made by one of the agents. We will ally with the red side and define a cost function according to the red agent's performance. The agent will attempt to maximize its reward at every move; for purposes of our optimization algorithm, we will model the problem in an inverse fashion, trying to minimize the cost in the objective rather than maximize a perceived reward. The objective function and the cost function are define as in the aforementioned value space approximation expression Eq. (3), where the individual stage costs are represented as $g_k(x_k, \mu_k(x_k), w_k)$ These costs are specified by an artificial score which we calibrate, and that quantifies the level of progress made by the red agent.

gN={0}

$g_k(x_k, \mu_k(x_k), w_k)$={

- 1,000,000 if the red agent has 4 checkers in a row
- 1, if the agent filled three spots, and the remaining spot is empty (the agent wins if it fills in the empty spot
- 100, if the opponent filled three spots, and the remaining spot is empty (the opponent wins by filling in the empty spot)

}

C. *Disturbances*

Equivalently, at each stage our system will undergo some disturbances. These disturbances are actually the actions of the yellow agent, our opponent. We cannot predict or model the distributions of these disturbances in an effective manner, hence we rely on heuristics and reinforcement learning methods to help us determine the best course of action.

w_k : random variable which changes the state in a seemingly unpredictable fashion.

D. *State Space*

Indeed, in order to understand how the disturbances affects our system, we must first define our state space. The state space that we choose depends on the game's features. In this instance, the state space is a representation of the board's geometry and of the occupied positions by the checkers of both players. We actually encode the state as a one-dimensional array, denoted x_k, by flattening the (6x7) 2D board into a (1x42) array. We differentiate the red and yellow agents' checkers by using different entry values: 0-valued entries for empty spaces, 1-valued entries for red player's checkers, and 2-valued entries for yellow agent's checkers.

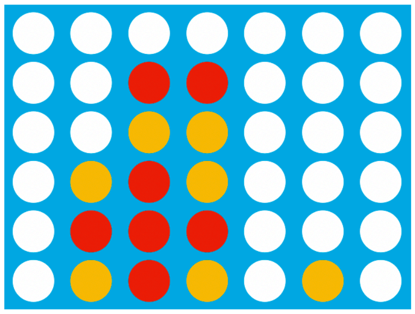


Figure 2: Encoding of state space

x_k=[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 2, 2, 0, 0, 0, 0, 2, 1, 2, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 2, 1, 2, 0, 2, 0]

E. *Policy Space and Controls*

Now, the policy of this game allows us to pick one out of seven available columns to drop one of our checkers. There are certain restrictions of course, such as the column not being completely full, but we can model the policy in very simple fashion, whereby the picked value corresponds to the desired column:

$\mu_k\ (x_k)$ ={1,2,3,4,5,6,7}

And the corresponding control is given by this small algorithm:

```
u = {
        N=xk
        u=zeros(N)
        I=N+
        for j=1:6
        if xk[i-7*j] ==0:
                u[i-7*j]=1
                break
}
```

F. Dynamics Function

Now that we have the state space, the policy and controls, and the disturbances, our **dynamics function** is a result of the interaction of all these quantities.

$$x_{k+1} = f_k(x_k, \mu_k(x_k), w_k), \forall k \in \{0, \ldots, N-1\}$$

such that

$$x_{k+1} = x_k + u_k + w_k$$

IV. Development of the Optimization Program

A. Game Environment and Agent Configuration

Kaggle provides a very neat platform with a basic match environment. The standard configuration is comprised of a configuration class, and an observation class. We instantiate objects from these classes and simulate games between agents. The agents evolve in the defined environment by accepting two object arguments and their corresponding fields:

- **configuration object** (rules of the game):
  - **_.columns** - number of columns in the board
  - **_.rows** - number of rows in the board
  - **_.inarow** - number of checkers in sequence needed to win
- **observation object**:
  - **_.board** - list representing the game board
  - **_.mark** - represents the piece assigned to the agent (either 1 or 2)

The _.board field in our observation object corresponds to the state at each stage of the game, x_k.

B. Evaluation Metrics

The outcome in a single game is not enough information to generalize a robust prediction of how well our agents perform. To get a better generalization, we compute a victory ratio for each agent, averaged over multiple games. For fairness sake, each agent is allowed to make the first move on half of the games played. The function that calculates the victory ratio is divided in two parts

- Test the agent against the baseline model: an agent which plays randomly
- Test the agent against a more complex model, defined by Kaggle (using "negamax" algorithm)

C. Random-choice Agent

The very first agent that we create produces a simple 'base policy'. Before delving into our DP agents, we want to establish a baseline agent that helps serves as a reference for performance. The random-choice agent has a gameplay policy whereby his choice of stage policy μ_k is determined by a random uniform distribution ~ Uniform(1,7). That is, from a set of admissible plays our base policy agent picks a column at random; game evolves through repeated simulations of this policy until the end-game condition is met.

1) *Random Model Code*

```
def random_agent(obs, config):
valid_moves = [col for col in range(config.columns) if
obs.board[col] == 0]
return random.choice(valid_moves)
```

The results are not very good, as expected. The agent is playing random moves, thus not a very smart policy.
This random-choice agent is our first agent, but its main use will be as an opponent to our more advanced RL agents. If they have no trouble beating this simple base policy, we could find some promising results.

D. One Step Lookahead Model

Next, we create an agent that uses heuristics for improved gameplay strategy. The heuristic we implement should be able to resemble some human's decision-making patterns. For example, some people attempt to forecast possible scenarios by accounting for the exact positions where pieces might have fallen after a few turns. This predictive power however has some limitations in both scope and precision. The heuristic we implement takes advantage of the Multistep Lookahead (or L-Step Lookahead) algorithm in order to find suitable gameplay moves.

Our general Lookahead approach uses a Deterministic Tree Search algorithm to list all possible outcomes given a current stage state. The agent then chooses the path that is most likely to yield a win, i.e. the one with the lowest cost-to-go $\tilde{J}_{k+1}$.

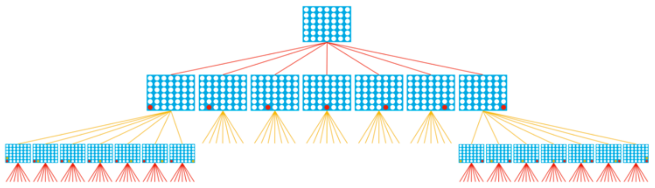


Figure 3: illustration of detailed tree search.

The combinatorics of the state space could require an excessive amount of computational power. Hence, we need to use fast approximation algorithms and consider incomplete trees. Our Lookahead function can define a shorter term strategy by computing costs to possible moves, only until a certain stage horizon.

In this section we implement the One-step Lookahead, where our agent relies on a strategy that sees 1 single step into the future, L = 1.

The approximate value functions for the One-step Lookahead is given by:
(5)

$$\tilde{J}_{k+1}(x_{k+1}) = \min_{u_{k+1} \in U_{k+1}(x_{k+1})} \mathbb{E}_{w_{k+1}} \left[g_{k+1}(x_{k+1}, \mu_{k+1}(x_{k+1}), w_{k+1}) + \tilde{J}_{k+2}(x_{k+2}) \right]$$

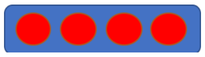
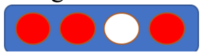
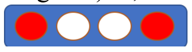
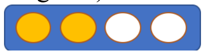
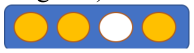
And the sub-optimal policy can be found using Eq. (4) and setting terminal cost $J_{k+2}(x_{k+2})$ equal to zero.

1) *Scoring Function*

Furthermore, we design a scoring function with the aim of quantifying the stage cost. The scoring function is implemented as exactly the negative of our stage cost. It is essentially a translation of our mathematical framework to the implementation for purposes of convenience:

$$\_score = -g_k$$

Following the scoring function logic, a state with high likelihood of achieving a win will be assigned a high score value. The algorithm will select the move that leads to the board which was assigned the highest score:


4 aligned red circles, meaning that we won the game: 10000

3 red and 1 free space on a line (vertical, horizontal or diagonal): 1,000

2 red and 2 free spaces on a line (vertical, horizontal or diagonal): 500

2 yellow and 2 free spaces on a line (vertical, horizontal or diagonal): -300

3 yellow and 1 free space on a line (vertical, horizontal or diagonal) : -1,000

Thanks to this heuristic function, our agent will be able to:
- detect and select a winning move.
- detect the opponent winning move and block it
- add disks on the same line, and/or diagonally

2) *One Step Lookahead Code*

```
def One_step_lookahead_agent(obs, config):
valid_moves = [c for c in range(config.columns) if
obs.board[c] == 0]
# Convert the xk list into a 2D array
grid = np.asarray(obs.board).reshape(config.rows,
config.columns)
# Assign scores to each new board
scores = dict(zip(valid_moves, [score_move(grid, col,
obs.mark, config) for col in valid_moves]))
# Select the moves that lead to the highest scores
max_cols = [key for key in scores.keys() if scores[key] ==
max(scores.values())]
# If many moves maximize the score, returns one randomly
return random.choice(max_cols)
```

3) *Evaluation*

We immediately observe our model wins 100% of the time against the random agent. Thus our algorithm checks the first element of the validation process we defined earlier. Now let's evaluate its performance against a more complex algorithm.

We simulate matches against the negamax algorithm, an agent that has already been implemented by Kaggle.

The results we get from these simulations average our agent a 55% victory ratio in favor of our agent. This tells us that there is still plenty of margin for improvement for our agents' design.

Method	*Win Percentage vs. Random*	*Win percentage vs. NegaMax*	*Average time per move*
One-Step Lookahead	*100%*	*55%*	*0.5 s*

E. Multistep Lookahead Agent

We wish to enhance our algorithm by expanding to a larger visibility horizon. Instead of forecasting 1 step, we want to account for 3, 4, 5 steps in the future. With a Multistep Lookahead, we can deploy a strategy that makes better informed decisions as it looks farther into the future. However, we have to balance the choice of L, as it can improve policy choice but at the expense of computational efficiency.

The expression for the general Multistep Lookahead was given in Eq(3), whereby the optimum is given by

(6)
$$\min_{u_k,\mu_{k+1},\dots,\mu_{k+l-1}} \mathbb{E}_{w_{k+1},\dots,w_{k+l-1}}\left[g_k(x_k,u_k,w_k)+\sum_{i=k+1}^{k+l-1} g_i(x_i,\mu_i(x_i),w_i)+\tilde{J}_{k+l}(x_{k+l})\right]$$

And again we set the terminal cost $\tilde{J}_{k+l}\,(x_{k+l})$ equal to zero.

For L=3, our program considers every possible board state after 3 stages (agent move, opponent move, and next agent move) and selects a policy based on the predicted cost-to-go as defined by the scoring function.

We will use the Minimax Algorithm for the opponent. This type of algorithm selects the policy which maximizes an agent's scoring function, while at the same time minimizing the opponent's score. The scoring function of the Minimax algorithm is a heuristic such as the one we presented earlier. Moreover, in order to reduce run-time, we use alpha-beta pruning. The pruning cuts off branches that need not be searched as the program has already found better possible paths.

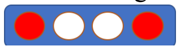
1) *Minimax Algorithm with Alpha-Beta Pruning*

```
def minimax(node, depth, MaximizingPlayer, alpha, beta):
    if node is a leaf node:
        return heuristic of the node
    if MaximizingPlayer:
        bestVal = - Infinity
        for each child node:
            value = minimax(node, depth+1, False, alpha, beta)
            bestVal = max(bestVal, value)
            alpha = max(alpha, bestVal)
            if beta <= alpha:
                break
        return bestVal
    else:
        bestVal = + Infinity
        for each child node:
            value = minimax(node, depth+1, True, alpha, beta)
            bestVal = min( bestVal, value)
            beta = min(beat, bestVal)
            if beta <= alpha:
                break
        return bestVal
```

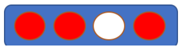
2) *Tuning the Heuristic Function*

We tune the heuristic function in order to make better strategies. For example, a very interesting strategy is the following:

- If at a special state x_k we can count at least 2 times the following scheme:



➔ A good policy would be to drop one more checker and turn these two schemes into two times the following:



Thus, at the next move, we will have two opportunities to win, and at the same time. The opponent can block one, but not two winning combinations… victory is guaranteed. Therefore, we need to increase the score value that corresponds to the last scheme

3) *Evaluation*

This 3-step Lookahead agent won 40 games out of 40 against the random-choice agent. Also, the agent has a victory ratio of 80% against the 'negamax' agent. However, we encounter a limit to this methodology, as the agent relies only on strategies thought of by the person who implemented the program. Hence the need for something more advanced arises, and will be explored in the last section section of our report.

Method	*Win Percentage vs. Random*	*Win percentage vs. NegaMax*	*Average time per move*
One-Step Lookahead	*100%*	*55%*	*0.05 s*
Multistep Lookahead	*100%*	*80%*	*0.5 s*

F. *Reinforcement Learning Agent*

Our aim for the last agent is to it uses RL methods. With these methods, our goal is to find the parameters that optimize certain parametric approximation architectures, and as a consequence, we minimize the agent's cumulative cost. In this case, the approximation architectures are modeled as a Neural Network. Agents built in previous steps are used as opponents and as source of learning for the RL agent.

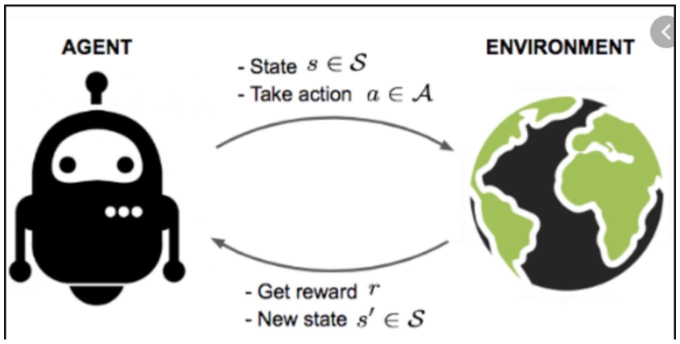

AGENT
ENVIRONMENT
- State $s \in \mathcal{S}$
- Take action $a \in \mathcal{A}$
- Get reward r
- New state $s' \in \mathcal{S}$

Figure 4: Learning of agent from environment information

Our chosen method is a model-free type of case, in the sense that we do not know the probability distributions of the variables that conform the agents. More specifically, we need to rely on a Neural Network with Proximal Policy Optimization to compute expectations.

Before we start delving into the specific implementation, let's first declare some variables that will aid us in our model formulation. One such variable is called the Q-factor, and its formulation is expressed as:

(7)

$$Q_k^*(x_k,u_k) = \mathbb{E}_{w_k}\left[g_k(x_k,u_k,w_k) + J_{k+1}^*(f_k(x_k,u_k,w_k))\right], \forall k \in \{0,...,N-1\}$$

The corresponding cost-to-go is then:

(8)

$$J_k^*(x_k) = \min_{u_k \in U_k(x_k)} \{Q_k^*(x_k,u_k)\}, \forall k \in \{0,...,N-1\}$$

The appropriate policy $\mu_k$is found again using Eq (4)

1) *Reward System*

Similar to the scoring function we defined for the Lookahead approaches, we set up a reward function for the neural network. The idea of the function is that after each move, the system will be rewarded some points so it can learn the good moves and the bad moves. Again, this reward is the negative of the cost for the purposes of the coding implementation. Indeed, one could present an optimization program of the reward maximization form and adapt the rest of the algorithms:

(9)
$$Q_k^*(x_k, u_k) = \mathbb{E}_{w_k}\left[r_k(x_k, u_k, w_k) + J_{k+1}^*\left(f_k(x_k, u_k, w_k)\right)\right], \forall k \in \{0, \ldots, N-1\}$$

(10)
$$J_k^*(x_k) = \max_{u_k \in U_k(x_k)} \{Q_k^*(x_k, u_k)\}, \forall k \in \{0, \ldots, N-1\}$$

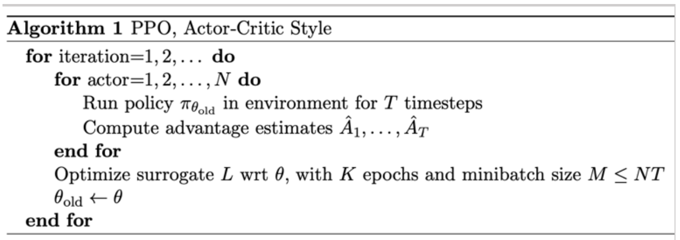
We defined the following reward system for a given reward, r_k:

- If after a move, the agent wins the game, it will be rewarded +1 point.
- If it loses, it will be rewarded - 1.
- If the move is invalid (playing in a full column for example), it will be rewarded -10 points.
- Else, the agent will be rewarded +1/42 (42 being the dimensions of the board = 6x7)

2) *Proximal Policy Optimization Algorithm for 2-layer NN*

The architecture of the NN has an output layer with 7 nodes, corresponding to the available policies. We initialize the script assigning random weights to each of the nodes. Then, as we train the network with the reward system, the PPO algorithm adapts the weights of the network with the objective of maximizing the cumulative reward.

Our goal is to produce superior results compared to the ones we found with the Multistep lookahead. So we decide to train the RL agent using the Multistep Lookahead agent that the program has built (the most efficient until now). In this way the neural network learns from relatively stronger opponents, and can therefore improve systematically at each iteration. The PPO algorithm is presented below:



```
Algorithm 1 PPO, Actor-Critic Style
for iteration=1, 2, ... do
    for actor=1, 2, ..., $N$ do
        Run policy $\pi_{\theta_{\text{old}}}$ in environment for $T$ timesteps
        Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
    end for
    Optimize surrogate $L$ wrt $\theta$, with $K$ epochs and minibatch size $M \leq NT$
    $\theta_{\text{old}} \leftarrow \theta$
end for
```

The next figure shows the rolling average of cumulative reward of the model, after a certain number of episodes.

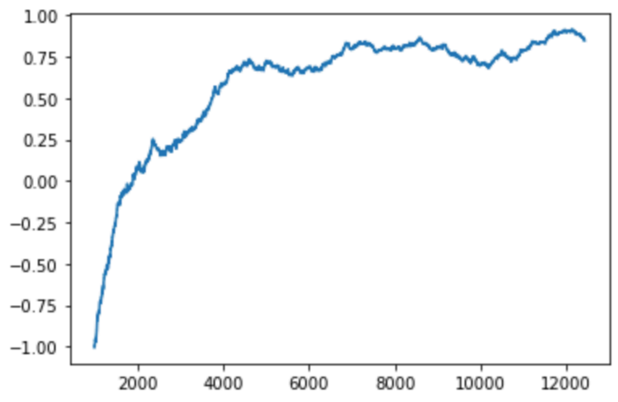


Figure 5: RL agent learning through time

As we can see, the model learns incrementally and its cumulative reward tends closely approaches 1. It learned how to select better policies by playing many times against its precursor agent.

V. Conclusion and Final Results

We decided to follow an incremental improvement of our algorithm. The first that we built is used as a baseline (the random agent). Then organically, we wanted an algorithm able to spot a winning move and thus win the game: this is the One-Step Lookahead. However, we know that humans try to predict a few moves into the future so as to deploy certain strategies. Thus we implemented the Multistep Lookahead to reproduce this ability to do strategies and take the one that would most likely lead to a victory.

Ultimately, for an algorithm to beat a human it must be able to execute complex strategies. Thus, in order to jump this frontier we used a Deep Reinforcement Learning agent. Our resulting agent plays games with older versions of itself, using a calibrated Neural Network architecture, and learns and develops more complex playing strategies. As of its current implementation and ranking, our final agent is capable of beating 80% of other smart agents implement by other users. There is still room for improvements and future work could explore the calibration of the neural network as well as its complexity.

BIBLIOGRAPHY

[1] Reinforcement Learning Applications by Yuxi Li, Medium.com

[2] Monte Carlo Q-learning for General Game Playing by Hui Wang, Michael Emmerich, Aske Plaat (https://arxiv.org/pdf/1802.05944.pdf)

[3] ConnectX by Kaggle, (https://www.kaggle.com/c/connectx)

[4] Proximal Policy Optimization Algorithms, by John Schulman et al, OpenAI, (2017)

[5] RL - Proximal Policy Optimization (PPO) Explained, by Jonathan Hui, (2018), Medium.com (https://medium.com/@jonathan_hui/rl-proximal-policy-optimization-ppo-explained-77f014ec3f12)

[6] Mastering the game of Go with deep neural networks and tree search, David Silver et. al

Appendix

My agents - ConnectX

May 15, 2020

1 My agents for Connect 4

```
[ ]: # This Python 3 environment comes with many helpful analytics libraries␣
      ↪installed
     # It is defined by the kaggle/python Docker image: https://github.com/kaggle/
      ↪docker-python
     # For example, here's several helpful packages to load

     import numpy as np # linear algebra
     import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

     # Input data files are available in the read-only "../input/" directory
     # For example, running this (by clicking run or pressing Shift+Enter) will list␣
      ↪all files under the input directory

     import os
     for dirname, _, filenames in os.walk('/kaggle/input'):
         for filename in filenames:
             print(os.path.join(dirname, filename))

     # You can write up to 5GB to the current directory (/kaggle/working/) that gets␣
      ↪preserved as output when you create a version using "Save & Run All"
     # You can also write temporary files to /kaggle/temp/, but they won't be saved␣
      ↪outside of the current session
```

```
[ ]: # 1. Enable Internet in the Kernel (Settings side pane)

     # 2. Curl cache may need purged if v0.1.6 cannot be found (uncomment if needed).
      ↪
     # !curl -X PURGE https://pypi.org/simple/kaggle-environments

     # ConnectX environment was defined in v0.1.6
     !pip install 'kaggle-environments>=0.1.6'
     from kaggle_environments import evaluate, make, utils
     import random
     import math
     import time
```

2 Create ConnectX environment

[ ]:
```
env = make("connectx", debug=True)
configuration = env.configuration
print(configuration)
```

[ ]:
```
print(list(env.agents))
```

3 Creating the Agents

[ ]:
```
def my_agent_one_step(obs, config):

    import random
    import numpy as np

    # 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_two = count_windows(grid, 2, mark, config)
        num_threes = count_windows(grid, 3, mark, config)
        num_fours = count_windows(grid, 4, mark, config)
        num_fours_opp = count_windows(grid, 4, mark%2+1, config)
        num_threes_opp = count_windows(grid, 3, mark%2+1, config)
        num_two_opp = count_windows(grid, 2, mark%2+1, config)
        score = 500*num_two + 6000*num_threes + 10000*num_fours -␣
↪15000*num_fours_opp - 10000*num_threes_opp - 300*num_two_opp
        return score

    # 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

    # 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)
```

```
[ ]: def my_agent_3_steps(obs, config):

    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

    # Helper function for score_move: calculates value of heuristic for grid
    def get_heuristic(grid, mark, config):
        num_two = count_windows(grid, 2, mark, config)
        num_threes = count_windows(grid, 3, mark, config)
        num_fours = count_windows(grid, 4, mark, config)
        num_fours_opp = count_windows(grid, 4, mark%2+1, config)
        num_threes_opp = count_windows(grid, 3, mark%2+1, config)
        num_two_opp = count_windows(grid, 2, mark%2+1, config)
        score = 500*num_two + 6000*num_threes + 10000*num_fours -␣
↪15000*num_fours_opp - 10000*num_threes_opp - 300*num_two_opp
        return score

    # 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):
        alpha = -100000
        beta = 100000
        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))
                alpha = max(alpha, value)
                if alpha >= beta:
                    break
            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))
                beta = min(beta, value)
                if alpha >= beta:
                    break
            return value
    # How deep to make the game tree: higher values take longer to run!
    N_STEPS = 3
    # 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)
```

4 Time to test our agents

[ ]:
```
# Agents play one game round
env.run([my_agent_3_steps, "random"])

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

[ ]:
```
def Victory_ratio(agent1, agent2, n):
    L = evaluate("connectx", [agent1, agent2], num_episodes=n)
    print(L)
    agent1_ratio = sum(L[i][0]==1 for i in range(len(L)))/n
    agent2_ratio = sum(L[i][1]==1 for i in range(len(L)))/n
    return agent1_ratio, agent2_ratio

Victory_ratio(my_agent_2_steps,"negamax",20)
```

[ ]:
```

```

Neural Network Agent RandomAgent

May 15, 2020

1 Neural Network

[ ]:
```
import os
import random
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

!pip install 'tensorflow==1.15.0'
```

[ ]:
```
import tensorflow as tf
tf.__version__
```

2 Connect4 Gym Class

[ ]:
```
from kaggle_environments import make, evaluate
from gym import spaces

class ConnectFourGym:
    def __init__(self, agent2="negamax"):
        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=(self.rows,self.columns,1),␣
 ↪dtype=np.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(self.rows,self.columns,1)
    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(self.rows,self.columns,1),␣
↪reward, done, _
```

3 Train to beat the random agent

[ ]:
```
# Create ConnectFour environment
env = ConnectFourGym(agent2="negamax")
```

4 Vectorized Environment

[ ]:
```
!pip install stable-baselines > /dev/null 2>&1
```

[ ]:
```
import os
from stable_baselines.bench import Monitor
from stable_baselines.common.vec_env import DummyVecEnv

# Create directory for logging training information
log_dir = "ppo/"
os.makedirs(log_dir, exist_ok=True)

# Logging progress
monitor_env = Monitor(env, log_dir, allow_early_resets=True)

# Create a vectorized environment
vec_env = DummyVecEnv([lambda: monitor_env])
```

5 Architecture

[ ]:
```
from stable_baselines import PPO2
from stable_baselines.a2c.utils import conv, linear, conv_to_fc
from stable_baselines.common.policies import CnnPolicy

# Neural network for predicting action values
def modified_cnn(scaled_images, **kwargs):
    activ = tf.nn.relu
    layer_1 = activ(conv(scaled_images, 'c1', n_filters=32, filter_size=3,␣
↪stride=1,
                         init_scale=np.sqrt(2), **kwargs))
    layer_2 = activ(conv(layer_1, 'c2', n_filters=64, filter_size=3, stride=1,
                         init_scale=np.sqrt(2), **kwargs))
    layer_2 = conv_to_fc(layer_2)
    return activ(linear(layer_2, 'fc1', n_hidden=512, init_scale=np.sqrt(2)))

class CustomCnnPolicy(CnnPolicy):
    def __init__(self, *args, **kwargs):
        super(CustomCnnPolicy, self).__init__(*args, **kwargs,␣
↪cnn_extractor=modified_cnn)

# Initialize agent
model = PPO2(CustomCnnPolicy, vec_env, verbose=0)
model2 = PPO2(CustomCnnPolicy, vec_env, verbose=0)
```

6 Train the agent

[ ]:
```
# Train agent
model.learn(total_timesteps=100000)

# Plot cumulative reward
with open(os.path.join(log_dir, "monitor.csv"), 'rt') as fh:
    firstline = fh.readline()
    assert firstline[0] == '#'
    df = pd.read_csv(fh, index_col=None)['r']
df.rolling(window=1000).mean().plot()
plt.show()
```

7 Agent in the format of the competition

[ ]:
```
def agent1(obs, config):
    # Use the best model to select a column
    col, _ = model.predict(np.array(obs['board']).reshape(6,7,1))
    # 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])
```

7.1 Game round against the random player

[ ]:
```
# Create the game environment
env = make("connectx", debug=True)

# Two random agents play one game round
env.run([agent1, "random"])

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

[ ]:
```
def Victory_ratio(agent1, agent2, n):
    L = evaluate("connectx", [agent1, agent2], num_episodes=n)
    print(L)
    agent1_ratio = sum(L[i][0]==1 for i in range(len(L)))/n
    agent2_ratio = sum(L[i][1]==1 for i in range(len(L)))/n
    return agent1_ratio, agent2_ratio

Victory_ratio(agent1,"negamax",2)
```