Reinforcement Learning in a ConnectX Competition

Udacity Machine Learning Engineering Nanodegree Program Capstone Project

David Abelin ~ February 16, 2021

Introduction

As laid out in the Capstone Proposal, the aim of this project was to train a neural network model in a reinforcement-learning environment sufficiently for it to perform well in the real-world ConnectX competition at Kaggle.

To that end, the plan was to:

1. design a competitive "heuristic" (non-NN) algorithm to play ConnectX
2. find the best hyperparameters for a reinforcement-learning environment and train the RLNN model by letting it "explore" the RL environment in play against the most competitive heuristic agent
3. evaluate the model:
    a. in play against various agents
    b. against a set of "perfect" moves
4. wrap the model up in a submittable form and submit it to the ConnectX competition
5. evaluate the model against the final metric of its placement on the scoreboard

As it turned out, only the first three steps were completed successfully. Nevertheless, for completion of this Project, I will describe what worked, and what didn't work, and offer possible paths to explore going forward.

I put a lot of effort into getting this project to work. Trying to write that failure up has been difficult. Finally, I have decided to just submit what I have, and go with that. Thank you for your understanding about that.

The Competition

The ConnectX competition at Kaggle has been running for over a year. The competition is of the classic game *Connect 4* and consists of direct play between "agents" in an environment given by Kaggle. Agents are defined as methods which take two variables about the state of the game, and return a single digit [0,6] representing the choice of which column to play next. Agents are submitted in the form of an executable python file.

As outlined in the project proposal, this project requires two types of game-playing agents. First there are the “classic” agents which in this case will all be implementations of the minimax algorithm with alpha-beta pruning (pseudocode shown[1]) along with a set of heuristics. These will then be used to train the second kind: these will be the neural-network models to be trained with Reinforcement Learning.

Heuristic Agents

A common computational approach[2] to two-person gameplay is to implement some version of the *minimax*[3] algorithm. For this project, all the non-neural-network agents rely on this algorithm to decide which move to make next, given its arguments representing the current state of play. These agents also use a variety of “heuristic” measures to assist in computing that response. I will therefore refer to these as *agents*, to distinguish them from *models*.

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            α := max(α, value)
            if α ≥ β then
                break (* β cutoff *)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            β := min(β, value)
            if β ≤ α then
                break (* α cutoff *)
        return value
```

The Reinforcement-Learning Environment

Game agents driven by reinforcement-learning models trained in a “stable-baselines” environment (as initially defined in the Kaggle course *Intro to Game AI and Reinforcement*) against one or more of the best-performing heuristic agents.
The RL models will be defined using a slightly modified version of the custom environment given in that notebook.[4]

---

[1] Russell, Norvig (2003)
[2] <<Citation needed>>
[3] Russell, Norvig (2003)
[4] Cook (2019)

Environment predefined at the Kaggle competition:

```
class ConnectFourGym:
    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=(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, step_reward, done):
        gridsize = self.rows*self.columns
        if step_reward == 1: # The agent won the game
            return 1
        elif done: # The opponent won the game
            return -1
        else: # Reward 1/42
            return 1/gridsize

    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, step_reward, done, _ = self.env.step(int(action))
            reward = self.change_reward(step_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, _
```

The idea behind Reinforcement Learning is to let agents “explore” a training environment, gradually learning how to behave in it by making guesses and receiving feedback about those guesses via a pre-defined reward system.

Initially, the reward scheme was given as: +1 point for making a move that wins the game, -1 point for making a losing move, 1/42 points for making any move that does not end the game, and -10 points for making an invalid move. Many other combinations of these values were tried, but to no great effect (see Table **X**).

Table X Hyperparameters tried:
A: 10/42, -100/42, 1/42, -10
B: 1/42, -100/42, 1/42, -420/42
C: 1/2*42, -210/42, 1/42, -420/42
D: -1/42, -300/42, 1/42, -420/42
E: -1/42, -300/42, 2/42, -420/42
F: -50/42, -300/42, 1/42, -400/42
G: -200/42, -300/42, 1/42, -400/42
H: -1/42, -200/42, 2/42, -400/42
J: 3/42, -42/42, 1/42, -420/42
O: 1, -1, 1/42, -10

After training, these models were evaluated in two ways: first, again by direct game-play with other game agents; and second, against a set of “perfect moves” generated by another player.[5]

It would be useful to have a relation for comparing performance during training to evaluation after training for the original scoring scheme [1, -1, 1/42, -10] to compare with the other. This relation can be worked out in general as:

Win percentage = (average score + (1 - **s** / 42)) / 2

Where:

average score = wins - losses / n, and:

n is the number of games played,
s is the number of steps for each game,

[5] Cnudde (2020)

s = s / n is the average steps per game
game score = **s**/42 + {-1:lose, 1:win}

Implementation

Submission.py -- quicklook_submit.py
processing steps: pre, refine, post

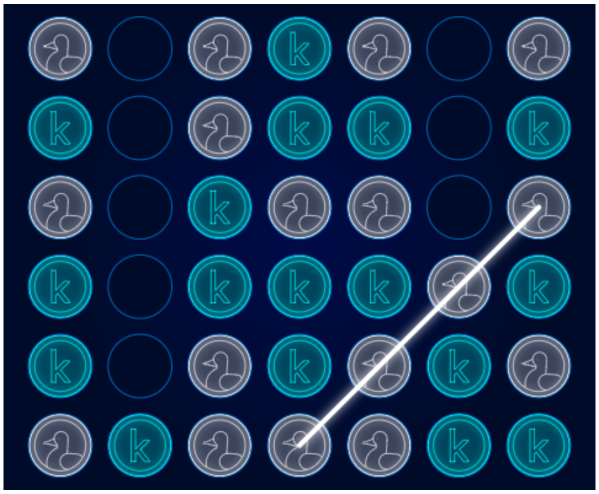
PPO1 and A2C models tried.

The most important aspect of implementation would have been the incorporation of a trained model into a ConnectX agent ready to play in the competition. This would require storing the model's parameters as a variable in the definition of the agent itself; for competition, only what can be put in the definition can be used as an agent, and there is no access to external storage once submitted. One possible way around this would be to store the parameters as a string, write that string to a file from within the definition, and then read those parameters normally with the **load** function for PPO1/A2C models.

Results

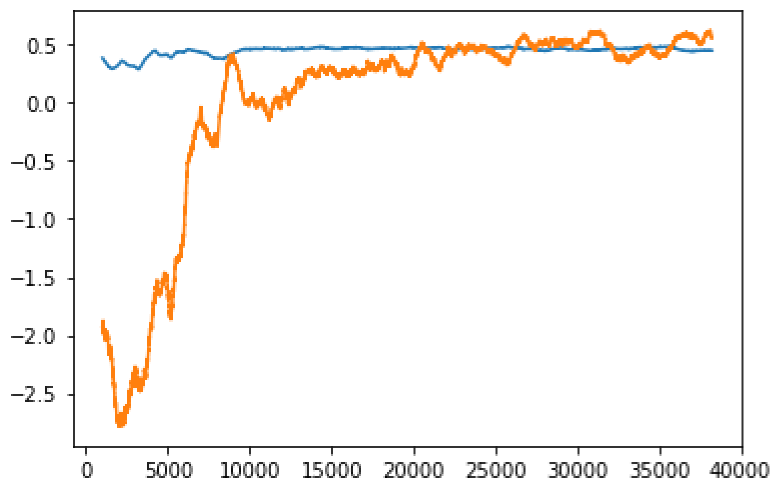
See also Appendices.

tables and graphs, interpretation: validation and justification
<screenshot scoreboard>



For the traditional heuristic models, the evaluation will be how well they perform in direct competition against other agents. The most successful of these models will then be chosen to train the RL models.
Best performing heuristic agent:
https://www.kaggle.com/davidabelin/test-agent?scriptVersionId=47506751

Training:



Learning curve screenshot

<average reward vs. average wins>

Table 2. Score and Rank of the three best performing agents

Agent	**Game Score**	**Rank in Competition**

Table 3: Heuristic Agents and the Perfect Move Scores

Agent	**Perfect Move Score**	**Good Move Score**
test_agent_v9	0.681	0.884
scoring test_agent_v6	0.68	0.886
heuristic	0.667	0.889
quick_look	0.65	0.863
strong_coeffs	0.639	0.856
quick_pick_submit	0.642	0.881
deep_lookahead	0.632	0.884
standard		

getWinPercent: 0.9236

After 110 games:
Agent 1 Win Percentage: 0.2091
Agent 2 Win Percentage: 0.7909
Number of Invalid Plays by Agent 1: 0
Number of Invalid Plays by Agent 2: 0
Total time taken: 533.2 seconds
Time taken per round: 4.8 seconds

Analysis, Evaluation, and Conclusion

In the end, what was accomplished was not even in the machine learning part of the project.
It remains unclear what went wrong with the reinforcement learning models and their training environments. No arrangement of metaparameters, design, or length of training had any effect on their performance outside the training environment.
I have spent a lot of time trying to figure this out, and it has been difficult to put this all together and submit it.

References:

Cnudde, Peter (2020), "Scoring ConnectX Agents"

Cook, Alexis (2019), Intro to Game AI and Reinforcement Learning Cf. notebook: Deep Reinforcement Learning

Russell, Stuart J.; Norvig, Peter (2003), Artificial Intelligence: A Modern Approach (2nd ed.), Upper Saddle River, New Jersey: Prentice Hall, ISBN 0-13-790395-2 [Alpha-beta pruning pseudocode]