1 The Question

Connect 4 is an intermediately-hard game. It is not as easy as tic-tac-toe, but not as hard as chess. On the classic 6 by 7 grid, there are over 4 trillion game states. Can we teach our AI agent to play Connect 4 in the same way DeepMind taught AlphaZero to play chess?

2 The Data

Our data will be generated in real time. More specifically, we will simulate games between two agents over x many games. The dataset will be comprised of the sequence of board states, the winning board, and the winner. We will then train our agent on these games.

3 The Approach

We will use a Keras Neural Network to train our agent.

4 Motivation

Since Connect 4 is not too simple, but also not overly complicated, it is a good introduction to learn neural networks in a two-player game environment.

5 Requirements

We will use Python version 3.6.9. We will also use Keras (version 2.4.3) and Tensorflow (version 2.3.1). These can be retrieved with `pip install keras` and `pip install tensorflow`. To install a specific version, append the version number, e.g. `pip install tensorflow==2.3.1`.

6 Setup

We have some static variables to encode the board and layout. These can be found in `const.py`. Next, we need to program the connect 4 game itself. The actual game and methods for the game can be found in `game.py`.

1. `reset_board:` generates empty board of size `NUM_ROW` by `NUM_COL`
2. `get_available_moves:` finds all legal moves on the board
3. `get_game_result:` finds whether the board has a winner, draw, or continue play
4. `move:` places a piece on the board

Next, we want to setup our players. There are 3 options:

- **random** - selects a random move out of the available moves
- **showstopper** - basic greedy. Will prioritize dropping on columns/rows of 3 to win and stopping columns/rows of 3 of the opponent. Otherwise, it will flip a coin. If it is tails, it randomly selects a move (this is to add entropy). If it is heads, it will play on the highest column.

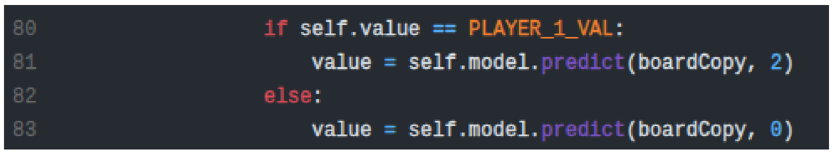
- **KerasNN** - uses the trained model to predict the best move.

More playstyles can be added by adding an `elif` with a label and passing in the strategy in `main.py`.

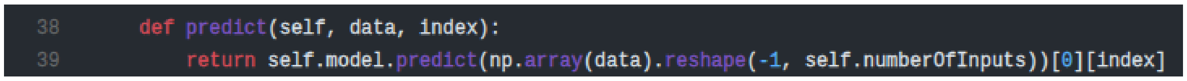
Next, we want to setup a game controller that will help us simulate games. The controller has a `play_game` method, which has the two agents play a game until one wins or there is draw. The `simulate_games` method is for statistics in recording wins, etc. The training history comprises of who won and the board, which will be used to train our model later.

The KerasNN is trained in `model.py`. We will have `NUM_ROW` by `NUM_COL` many inputs, as that is how many possible moves there are. Each of these map to 3 outputs: win, lose, or draw. Between them are 2 Dense layers of size 42 (layer numbers, types, or sizes can be adjusted - see what works best for you)

The training will map each board to who won, and hopefully, when it performs a fit, it will develop a strategy to determine good moves given game states. The KerasNN agent will call `predict` every time it needs a move. The idea is that Player 1 will want to maximize their probability of winning (given by output 2) and Player 2 will want to maximize their chance of winning (given by output 0).



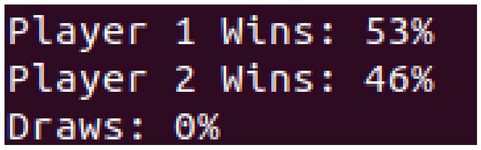
```
                if self.value == PLAYER_1_VAL:
                    value = self.model.predict(boardCopy, 2)
                else:
                    value = self.model.predict(boardCopy, 0)
```



```
    def predict(self, data, index):
        return self.model.predict(np.array(data).reshape(-1, self.numberOfInputs))[0][index]
```

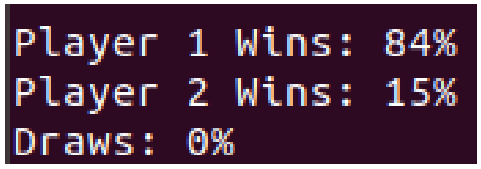
7 The Main

`main.py` will be used to put everything together. To get a baseline, we pitch two random agents against each other. We hover around Player 1 having a slight edge.



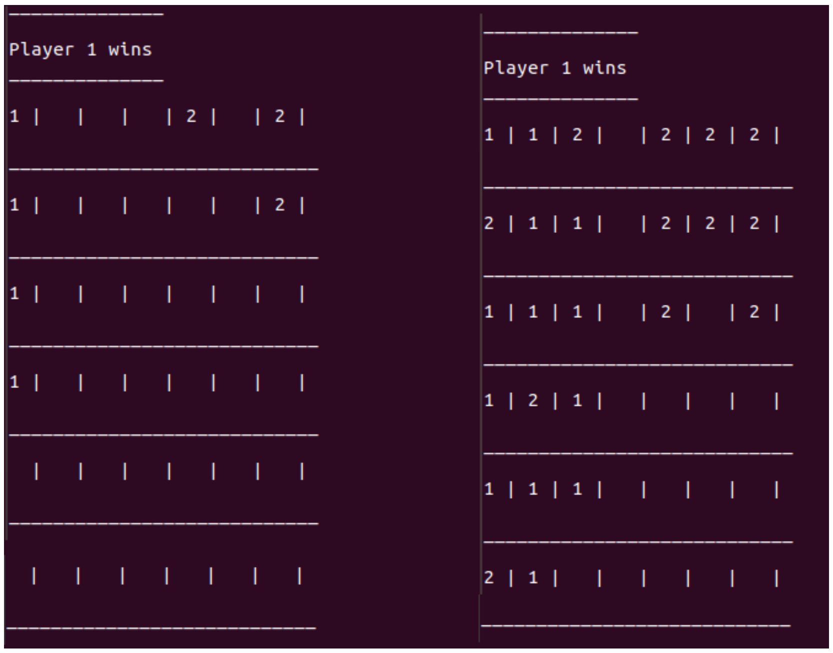
```
Player 1 Wins: 53%
Player 2 Wins: 46%
Draws: 0%
```

Now, we will train our model to these games with batch sizes of 21 and 100 epochs. We will then pitch our trained model against the random agent and see how it performs.



```
Player 1 Wins: 84%
Player 2 Wins: 15%
Draws: 0%
```

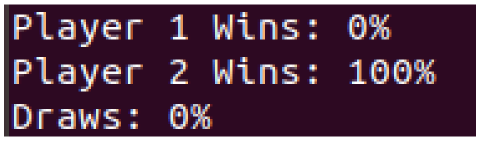
Clearly, it performs much better! However, it we take a look at some of the games it wins (boards go up), we find that it has picked up some undesirable behaviors. As the training data is based off of agents who play randomly, it realized it has a good chance of simply stacking four pieces in the same column. If the agent is player 1, then it has a $(6/7)^3$ chance of succeeding, i.e. so long as the random agent does not drop it in their column before hitting 4 in a row.



```
--------------
Player 1 wins
--------------
1 |   |   |   | 2 |   | 2 |
----------------------------
1 |   |   |   |   |   | 2 |
----------------------------
1 |   |   |   |   |   |   |
----------------------------
1 |   |   |   |   |   |   |
----------------------------
  |   |   |   |   |   |   |
----------------------------
  |   |   |   |   |   |   |
----------------------------
```

```
--------------
Player 1 wins
--------------
1 | 1 | 2 |   | 2 | 2 | 2 |
----------------------------
2 | 1 | 1 |   | 2 | 2 | 2 |
----------------------------
1 | 1 | 1 |   | 2 |   | 2 |
----------------------------
1 | 2 | 1 |   |   |   |   |
----------------------------
1 | 1 | 1 |   |   |   |   |
----------------------------
2 | 1 |   |   |   |   |   |
----------------------------
```

Thus, if we again train our agent on the two random agents, then pitch it against our showstopper agent, we see that our KerasNN gets demolished.



```
Player 1 Wins: 0%
Player 2 Wins: 100%
Draws: 0%
```

8 For The Reader

As we can see, our agent is effective against the random strategy, but loses even to a simple greedy strategy. The challenge for the reader is to develop an agent that can better help the KerasNN learn how to play Connect 4. Additionally, the reader can tinker with certain aspects of the training phase, e.g. changing the layer numbers/types, and see how these affect the KerasNN agent's performance. The challenge is to beat the simple greedy showstopper agent provided.