Machine Learning Engineer Nanodegree

Capstone Proposal

Armando Suarez
February 22, 2017

1. Domain Background

Conniption is a modified connect-four game, in which the player has the possibility of flipping the board upside down. When a flip occurs, the chips falls down creating new combinations, and in some cases a winning solution. In each player's turn the player decides to flip or not, place and again to flip or not. Each player is allowed to flip four times during a game and no consecutive flips between players are allowed. The rest of the game remains the same, a user place a chip from above in a 7x6 grid and finished his turn. A second player repeats the process in his turn. The winner has to link 4 consecutive chips in any possible direction.

As part of a class project, we created a command line python based version of the game that was implemented using a minimax algorithm with alpha-beta pruning, called conniption[1]. In such implementation, 5 different evaluation functions were created (Random, Sols, Cells, Hybrid, Flip), please refer to the documentation for more detail information. My personal motivation for this project, is to take something that I previously created and expand to improve it.

2. Problem Statement

In this project we will use reinforcement learning (with the option of adding deep learning) to make and agent play conniption, with the final goal to defeat the agents build upon mini-max with alpha-beta pruning.

a. **Task:** Playing Conniption
b. **Performance:** Percent of games won against other players. Due to the characteristics of the game it matters whether you start as the player one or player Two.

The main goal will be to build a reinforcement learning agent that will always maximize the next movement in order to obtain the best possible move.

3. Datasets and Inputs

The dataset will be constructed by playing the minimax agents against itself. This will allow us to implement reinforcement learning. Since we have different evaluation

[1] https://github.com/YeahHmm/Conniption

functions, all those functions will be passed to the agent during the training face in order to obtain different results. The game state is defined under the SystemState class in the resource module. The grid is sense by evaluation function as a 7x6 grid, where 0 represents an empty cell and 1,2 represent a player.

4. **Solution Statements**

In order to tackle the above problems we will use Q-learning, following the reinforcement model to learn from the evaluation functions by just playing the game. This will provide the algorithm with domain knowledge in order to tackle the next action. The board is evaluated by a passing a system state to the evaluation function, the function then provides a grading of the state with respect to its characteristics. A positive value will indicate an advantage for player one while a negative one will indicate an advantage for player 2. The minimax algorithm takes into account those values while creating the minimax tree. The values returned by the evaluation function will be used to attach rewards to the agent.

5. **Benchmark Model**

The Q-Learning agent will be tested based on its performance against the following agents that contain the following characteristics.

a. **Random Agent,** returns a random number from 0 to 10 as the evaluation function number.
b. **Sols Agent,** there is a specific set of winning positions across the board. Sols iterates over that set, identifying those solutions that have exactly one player's tile(s) inside of it. For each of these solutions, it count the number of tiles inside and assigns a weight to that solution accordingly. The weights are defined as 1 for one tile, 4 for two tiles, 16 for three tiles, and 1024 for four tiles. Winning states are assigned the highest priority. Each player's score is the sum of its weights. Player 2's score is always subtracted from Player 1's score.
c. **Cells Agent,** returns a raw summation of weights placed on each tile on the board. Each weight is calculated by the number of solutions possible from a single position on the board. For example the corner tiles each award a value of 3 since victory and only be achieved in three separate cases (once horizontally, once vertically, and once diagonally). The center positions ([4,3] and [4,4]) on the other hand are rated much more highly with a total of 13 possible solutions, which would return a value of 13.
d. **Hybrid Agent,** returns the sum of both the Sols and Cells evaluation functions. During the start of the game, Cells accounts for most of the total evaluation of the two functions. This is because of two reasons: first at the

start of the game, the board is generally empty thus there are many chains of tiles which results in a lower return from Sols. Secondly, the value returned by Sols happens to be exponential which means that single tiles, or chains of two matter much less than chains of three or four, which allows for the value Cells to be more influential in the beginning of the games. One last observation is that during the middle to late game, Cells also serves as a tiebreaker for Sols, generally choosing the center most tiles when all else is equal in Sols.

e. **Flip Agent,** when testing each of our evaluation functions, over time we noticed that flipping the board proved to be one of the most powerful mechanics in the game. Often times a player could flip aggressively whenever an advantage was present to quickly snowball to victory. However, we observed that our evaluation functions would greedily flip at any point it would improve the board state. Since flips are a finite yet powerful resource, we considered adding a cost to flips in order to encourage conservative play. The system state records every flip that is made and subtracts the overall value returned from the evaluation function by 10 for every flip made. This resulted in the program deciding to only flip either when it greatly increased the board state (to the point where victory would be assumed) or defensively to prevent the opponent from winning. As a result, we saw incredible improvements to the win percentages Gunter et Al, 2 when Flip was applied to Player 2 against the previous evaluation functions. When applied to Player 1 Flip as a perfect win record with zero losses and zero ties. More detail of each matchup will be discussed in the results.

6. **Evaluation Metrics**
    a. **Winning Percentage over a 100 games agents all 5 agents.**
        i. Starting as player 1
        ii. Starting as player 2

7. **Project Design**
    a. **Machine Learning Design**
        i. **Training experience.** Agents playing against each other.
        ii. **Learning Algorithm.** Q-learning. Model will be feed information from different evaluation functions to update the state information.
    b. **Project Details**

Since it's creation, Conniption was created with the capability of being extensible in the future. There are four main components to the game.

i. Resource, is the foundation of the program, providing supporting data types for use in the game's main loop and in the AI's searching algorithms.
ii. Game, contains high level classes implementing the algorithms and methods to correctly interact with game states.
iii. Evaluation, simply contains the various functions used by configurable AI.
iv. Main, entry point of the application where the primary game loop is located.

From a high level description, within the main module the type of players for the simulation are declared. Afterwards, the game enters in a loop of a determined amount of the simulations to be performed. Information about the games played, such as average number of turns and win-loss-tie information is stored available for being saved. Then the game is updated by asking the player class to select the next move. The player class is intended for extension, which then is used to create the AI class.

```
class Player:
    def __init__(self, name):
        self._name = name
class AI(Player):
    def __init__(self, name, evalFunc, max_depth=1, tieChoice=None):
        self.evalFunc = evalFunc
        self.tieChoice = tieChoice
        self._max_depth = max_depth
        super().__init__(name)
```

The general implementation will be the following. The learning aspect of the Q learning will be implemented in the main module. New flags will be added, to add a new case in which if the game is learning, it will run a specific number of times (based on the decay function). When the learning process is done, the information will be passed into a new player class derived from the above mentioned. The QL (Q-Learning) player class will take the learned model and will then be the one in charge of decision taking when playing against other models.