# Machine Learning Engineer Nanodegree

# Reinforcement Learning in Conniption

Armando Suarez
March 28, 2017

## I. Definition

### 1. Project Overview

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 benchmarks section for a discussion on the agents. 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 to make an agent play conniption, with the final goal being 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. We will analyze the results from using a predictive model (minimax algorithm) against learning through repetition (reinforcement learning). Connect 4 is a deterministic solved game in which

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

under optimum play from both players, player one will always win. We will explore if the same behavior is translated into Conniption and the influence it has on the agents.

3. **Evaluation Metrics**
    a. **Placement in performance against all 5 agents.**

    On the previous iteration of the game, the benchmark agents were evaluated against each other and the following ranking was reached:

    i. **Flip**
    ii. **Hybrid**
    iii. **Sols**
    iv. **Cells**

    This ranking was created by gathering the winning percentage over a 100 against each of the 5 different agents.

    i. Starting as player 1
    ii. Starting as player 2

    The new agents will be evaluated agents all agents and their performance will be measured in terms of their relative placement.

    This metric was selected since winning is the most important aspect of the game. A close loss is still counted as a loss, and we want to develop agents capable of winning agents the benchmark agents.

## II. Analysis

1. **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 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.

For a more detail explanation of the Q-Table representations, please refer to the part 3 of the Methodology section called *Q-Table Implementation.*

2. **Exploratory Visualization**

During the first iteration of the project, building the game and the minimax functions, we run all five evaluation functions against each other for a 100 games providing a total of 2,500 game simulations. We discovered that adding the flips to the game, did not modify the advantage the player one possesses over player two in the game. The evaluation functions *won all one hundred game* simulations against the other evaluation functions, excluding random, in all hundred games it played against the other. Random was able to beat the other three evaluation functions playing as P2 10 times out of 300 games representing 3.33 % of the times. This behavior can be explained by the Minimax algorithm assumption that the opponent moves will always be optimal and that the opponent behavior should be similar to its own, however our random player never quite followed this rule making moves that could be considered as ‘stupid’ or meaningless.

This specific behavior described previously can be seen where Random played as P1 against the other heuristics. As can be observed in the following graph, with the observed advantage of placing first Random was able to win 77 games out of 400 for an average of 19% of the total times, this is reflected in figure 1. Since our three main evaluation functions [Hybrid, Sols, Cells] won a 100% of the times playing against each other as P1, we decided to test which evaluation function behaves better in overall by taking into considerations the number of chips placed on the board. The assumption is simple, the higher the number of chips placed the better the evaluation function is as P2, making it harder for P1 to beat him. The results can be observed in figure 2.

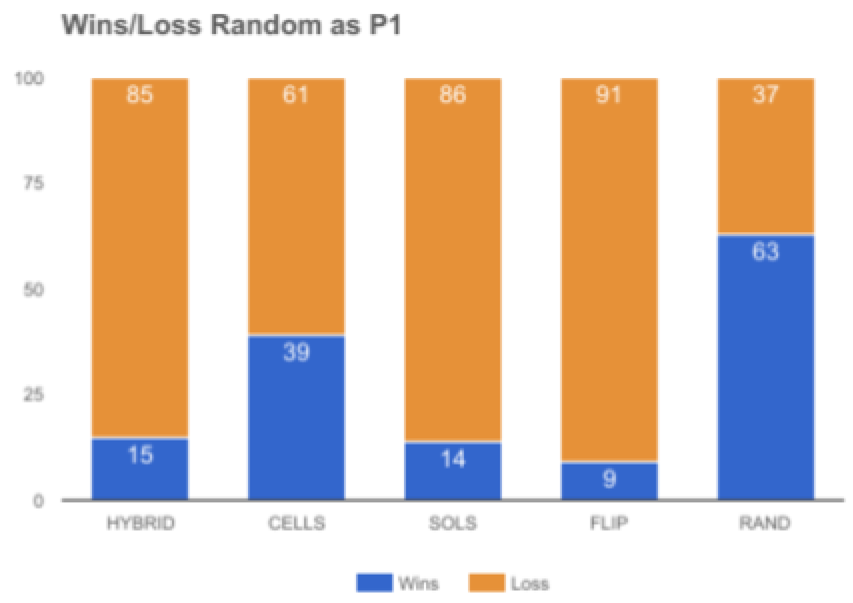

Figure 1: Random as Player 1 against all other users

The data presented above is the average of the number of plies taken across a 100 games. The game took an average of 12.03 moves across the 1,600 games. The Hybrid playing as P2 recorded a total of 17.73 moves per play, the best under the criteria previously analyzed. The lowest, as expected, was random with 8.67 and all the other ranged in between 10 and 12.

This data confirmed our assumption of having Hybrid as the best overall evaluation function. However, when playing human versus AI, and the AI playing as P2 it was fairly easy for the human to win due to a particular behavior of aggressive flipping. Our program did not place any specific weight to performing a flip, instead it analyzed all the possible moves and if the board with a flip presented a better option than the current one it will pick it every time. This was significant due to the fact that all the flips were performed early in the game and even in a situation in which it might not have been entirely necessary.

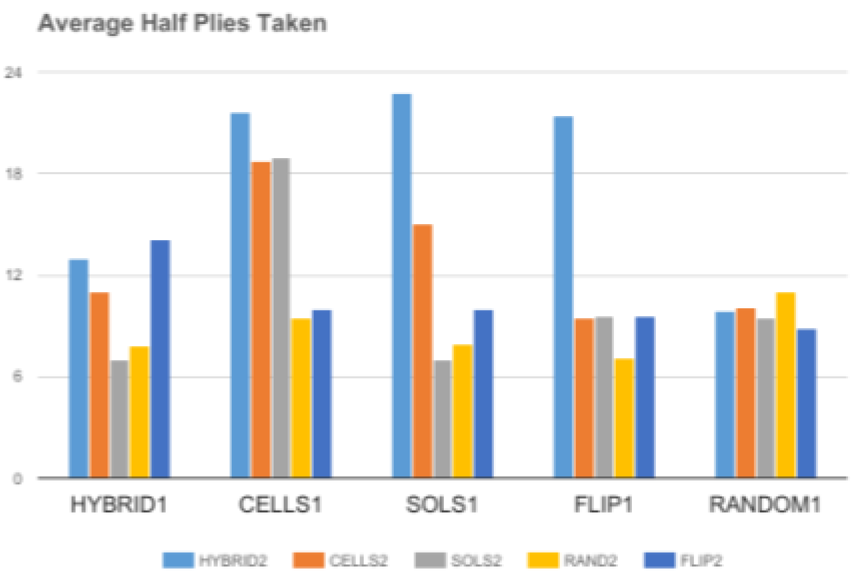

Figure 2: Average Half Plies taken by all agents against each other

We created a new heuristic called Flip, which takes the scored obtained by Hybrid and adds a bias to that score by calculating the number of flips left. The new behavior created tells the AI to perform a flip when it provides an advantage higher than the placed bias. After running the new set of tests, Flip behaved as expected as P1 and won all of its 500 games. Now let's analyze how it behaved playing as P2, and how well it.

The observed results are surprising. The Flip heuristic was able to defeat the other heuristics, excluding itself, 391 out of 400 times for a 97%. The nine losses accumulated against the random agent. Digging more into the data, it is even more surprising to find that all of the victories as P2 also came with the second lowest average of plies played with 10.5.

With this data we can reach two final conclusions:

- Under similar game strategies (heuristics) player one possess a great advantage and is ensured to win using optimal moves.

- The flipping behavior is a fundamental part of the strategy and dominating this, as player one, will almost guarantee a win. Flip as P1 has not be defeated by other heuristics or humans up to date. Undefeated 550-0.

3. **Algorithms and Techniques**

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 value 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.

Two different agents will be created, Qlearn and Minimax Qlearn, with the same underlying implementation. Both agents are following the same programing patterns and only differ in terms of the rewards and the specific values being passed to them. The agents differ in the following characteristics:

a. **QLearn:** The QLearn agent will get the rewards from the evaluation of the board after an action was taken, provided by the evaluation function. It evaluates the current state of the board without any prediction of future moves by the rival.
b. **MinimaxQ:** This agent gets the rewards based on the minimax search tree for the current board position. Meaning it uses future moves prediction to establish the optimal move for a specific state.

Under the above circumstances, and due to the fine-toned aspect of the minimax algorithms being used, the expected behavior will be for the minimaxQ agent to perform better. The other aspects of the QLearn algorithm will be implemented, such as epsilon and alpha. The initial decay function will be a simple linear implementation.

Q-learning aims to find the optimal action/policy for a finite Markov Decision Processes (MDP)[2]. From a high level perspective, the reinforcement learning is implemented in the following manner. A specific number of games will be simulated. This number of games is defined by two factors, the epsilon value and the tolerance. As long as the epsilon is bigger than the tolerance, the simulations will continue. Every time a new game is started, the epsilon value is updated by an epsilon decay function that is provided. The epsilon decay formula will ensure that in every iteration the epsilon value approaches the tolerance. In each specific game, when a state is encountered an appropriate entry in the Q-table is created. If the epsilon value is less than a number

[2] Markov decision process

randomly generated, then a random action is selected and the reward value for that action is saved. This is called the exploration factor. Else, the best available action is selected from the Q-Table.

```
new_q = (1 - rate) * old_q + (reward * rate)
```

The  import aspect of the learning is defined by the above formula. Where rate is a static value in between {0..1} and old_q is the value stored in the Q table for that specific state and action. The new_q value is stored again in the Q table after completed. The rate value simple defines the extent in which the new information will override the previous one.

## 4. Benchmark Model

The Q-Learning and the Minimax Q-Learning agents 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 can 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 due to 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.

## III. Methodology

### 1. Program Architecture

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

a. **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.
b. **Game**, contains high level classes implementing the algorithms and methods to correctly interact with game states.
c. **Evaluation**, simply contains the various functions used by configurable AI.
d. **Main**, entry point of the application where the primary game loop is located.

During the game, there are 3 decisions that should be taken before finalizing a player's turn: (flip or none), place, and (flip or none). To classify this player as a turn, we encapsulate this decisions into a half ply, as seen in figure 3. A ply is an instance of the minimax algorithm that indicates the turn of a player is completed.

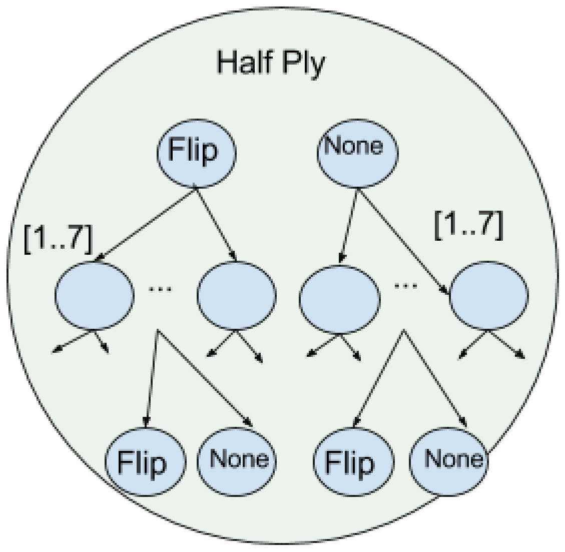

Figure 3: A half Ply node

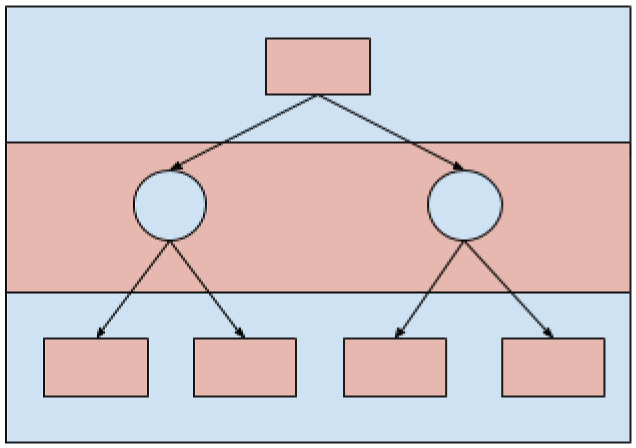

Figure 4: Minimax search on one ply. Squares represent a player, while a circle represent the opponent. It differentiates if minimizing or maximizing

We can observe our Minimax[3] implementation in figure 4. A node is encapsulated as a half ply, which follows the previously discussed inner implementation. After testing, a depth of 3 plies was defined as the more accurate depth without sacrificing accuracy. The same depth will be maintained for the reinforcement learning implementations.

## 2. Implementation

Based upon the four main models previously described, the following modules were added in order to implement the Q Learning implementation:

a. **Reinforce,** this module contains both player agents; QLearn, which extends from the Player class, and MinimaxQ, which extends the AI class.
b. **Simulator,** the simulator class manages the loop control for the Qlearn algorithm. It runs simulations until the epsilon value is less than the tolerance. When a game is completed, it resets all the values to the default values.
c. **MainQ,** this a function that resides inside of the *Main Module*. This function is intended specifically for training Q learning based models against all other agents. It allows for passing key parameters in order to modify the behavior of our agents. When just comparisons between agents is expected, the normal main() function should be used. The main function allows the addition of a human player against all other trained models.

[3] https://en.wikipedia.org/wiki/Minimax

Being more specific about the actual implementation of the QLearn agents, a specific state of a game is derived from the SystemState class which posses key characteristics necessary for updating the game actions. The game moves forward by creating a new instance of the System State with the new required positioning when game.update() is called.

```
class SystemState:
    def __init__(self, board=list(repeat([], 7)), prev_move=Move(), \
            player=0, num_flips=(0, 0), is_down=0, stage=0, num_placed=0):
        self._board = board
        self._prev_move = prev_move
        self._player = player
        self._num_flips = num_flips
        self._is_down = is_down
        self._stage = stage
        self._num_placed = num_placed
```

Inside of the reinforcement agents, once the new state is selected the {Qlearn, Minimax Q}.learn() function is called. This function, depending on the case, makes the appropriate calls to obtain the reward for the newly selected state. The value then is used to update the agent’s Q-table.

Due to the training time for all sets takes ours, the Q-table objects are stored into a pickle file, which allows for reuse of the train model under the main() method. This allows for faster testing and the flexibility of including automated game generations (the automation can be observed in the .sh files).

3. **Q-Table Implementation**

The Q-table keys are defined by using two aspects of the SystemState Class. A Key is stated the hash representation of (board, stage). Where the board represents where the pieces are positioned and stage (1..3) represent the stage of the half ply where the turn is located. Based on the representation then we can calculate the amount of states. The following calculations will give us the high upper bound.

```
States = (3[(p1, p2, empty)] ^ 42[board positions]) * 3[stages]
States = 9^42
States = 1.1972515e+40
```

The actual value is lower, since most cells could only be empty when the cells below them does not contain a chip. So a generalization will be to have those 7 possibilities extracted from the overall calculation. A more realistic bound would be:

```
 States = ((3[(p1, p2, empty)] ^ 7[board positions]) * 35) * 3[stages]
States = 98415 * 3
States = 295245
```

This should be a more accurate upper bound, those are the possible number of states if all the pieces are placed on the board. The game as shown before is finish in an average of 10 plies, so a realistic number of states used is smaller.

The inner dictionary of the Q-table is created based on the stage of the plie, with the following structures. If the stage is 1 or 3, assign the possible flip or none. If it is two, the seven rows to be placed are declared.

```
    self.def_dic_flip = lambda: { 'flip': 0.0, 'none': 0.0}
    self.def_dic_place = lambda: {0: 0., 1: 0., 2: 0., 3: 0., 4: 0., 5: 0.,
6: 0.}
```

So in order to update a value the next instruction will be called:

```
    self.Q[state.__hash__()][action] = new_q
```

Where the state._hash__() is the hashed value for (board, state) and the action is either {'flip', 'none} or {0...6} depending on the stage of the half ply. The following examples represent the 3 aspects previously described for the first half ply of the game, when no chips have been placed on the board.

- Stage 1:
    - `State:  -3381563282070318111`
    - `Action:  none`
    - `{'none': 0.0, 'flip': -10.0}`
- Stage 2:
    - `State:  -3381563282069235586`
    - `Action:  3`
    - `{0: 5.983976991608388, 1: 7.750213832728148, 2: 5.272478974909774, 3: 14.0, 4: 8.839534160994948, 5: 7.979202633953221, 6: 5.999997072513391}`

**4. Refinement**

The refinement aspect of the project was the most challenging part. Specially due to the difficulty for grading the performance of a specific agent in terms of wins/ losses. In order to determine if the agent is electing the optimal path, it was necessary to follow a manual tracing of the elections the agent was taking in terms of the game state.

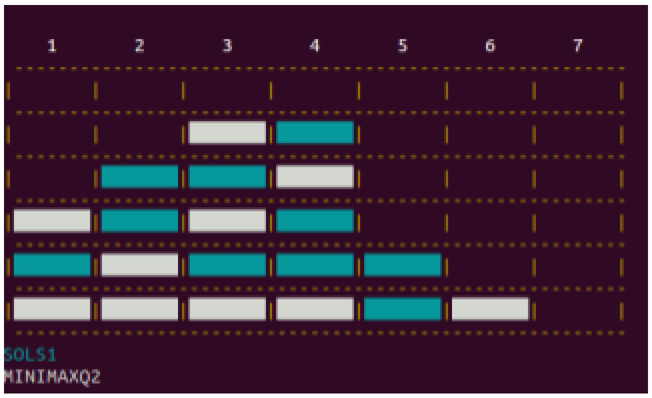

Image 1: A contested game with no flips remaining

At the beginning we started by implementing a linear epsilon decay formula, with an Alpha value of 0.5 and a tolerance of .05. This model was the less optimal. The number of trainings was not enough. Even after changing the the tolerance and the alpha, the model failed to generalize an optimal solution. Including the first key placement in column 4.

The cosine epsilon decay, fluctuated in between positive and negative values. This behavior was not acceptable since a negative value is always less than the decay. Using the absolute value made the function to fluctuate between 0.1 and 0.99 in iterative periods.

Finally, we arrive to the exponential epsilon decay function. It performed as expected and the following were the best values based on a specific agent.

| Agent | Epsilon Decay | Alpha | A | Tolerance | Trainings |
|---|---|---|---|---|---|
| QLearn | math.exp(1)^(-a*self.trials) | 0.45 | 0.004 | 0.001 | ~1727 |
| Minimax Q | math.exp(1)^(-a*self.trials) | 0.45 | 0.005 | 0.001 | ~1382 |

## IV. Results

### 5. Model Evaluation and Validation

Based on the selected models above, an automated script was used to train all the models against the proper evaluation functions. The resulting trained models were saved as a pickle file, and used posteriorly to obtain the following results.

#### 4.1 QLearn Agent

The QLearn agent was evaluated against all the benchmarks previously described, both as player one and player two for a total of 1000 different game simulations. As can be observed in figure 5, this agent was not able to win more than 50 games in any scenario, not even as player 1 or player 2. Overall it had a better performance as player 1, winning 17 times against the Hybrid (third best evaluation function) and 43 games against the Cell's agent. As player two, this particular agent was able to snatch 7 games from Cells acting as player 2, and surprisingly 49 from the Sols agent acting as player two. This indicates that the agent was not able to reach it’s main mission: to win more games than their adversaries. This behavior was expected, since the agent assigns rewards based on the specific board state while the minimax agents are predicting three plies into the future to select their own moves.

The above assumption will indicate that the agent was using a more defensive behavior, trying to block the attacks presented by the minimax agents. Attempting to prolong the game as much as possible. This can be seen by the number of plies taken during the games simulation (as mentioned above a ply is when both player 1 and player 2 complete their turns). The average plies taken across all simulations is 9.27, while Qlearn as player 1 averaged 13.91 and 15.36 plies as player 2. In the particular case of Qlearn 2 vs Sols 2, 19.57 plies were taken, which indicates how a highly contested game was split in the last games.

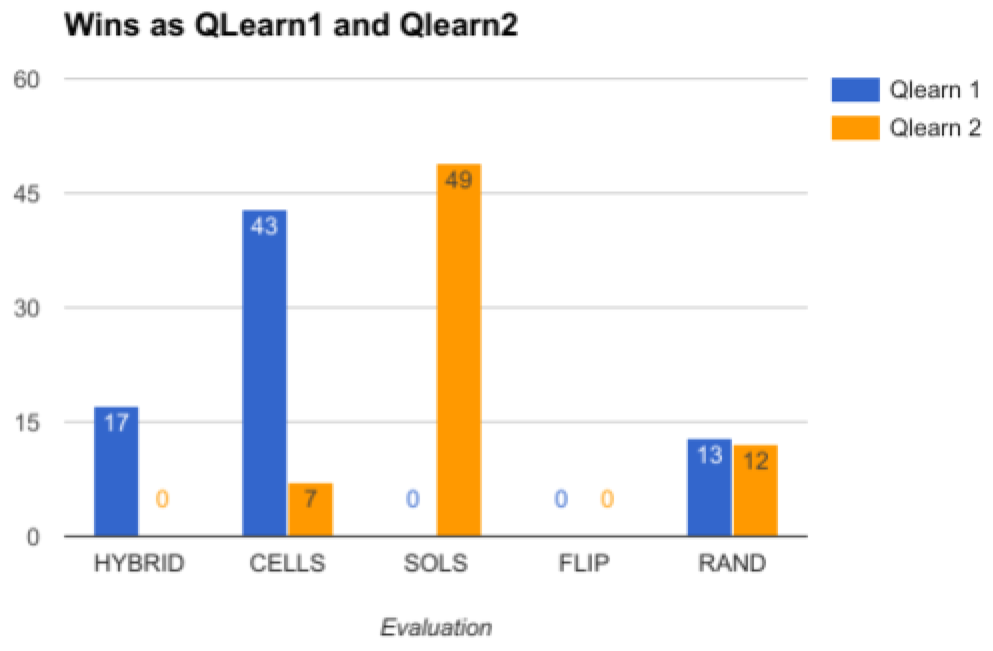

Figure 5: QLearn Agent Against All Others

### 4.2 Minimax Q

The minimax agent was also evaluated against all the other benchmarks for a total of 1000 different simulations, 500 as player 1 and 500 as player 2. As can be observed in figure 6, the minimax Q agent had a better performance overall than the previous QLearn agent. In this agent, the rewards are awarded based on the values that are returned from the minimax search with alpha beta pruning. Giving the agent better tools to create a winning strategy. Performing as player 1, the agent was able to defeat three of the main strategies (Cells, Sols and Flip) every single time. This means that the minimax Q agent was able to find the underlying path for a consistent winning strategy. In the previous iteration of the project, Flip was undefeated as player one and won 391 times out of 400 for a 97% (against all agents but itself) as player 2. Those 9 losses came against random. MinimaxQ is the first agent capable of defeating the Flip agent performing as player 2, and it was able to do it with a perfect score. A remarkable achievement.

Performing as player 2, it was not able to generate any considerable achievements for Hybrid, Cells or Flip with a combined of 0 wins in those 300 simulations. However, it was able to defeat the Sols function with a perfect score. In order to defeat Sols the game took an average of 21 plie.

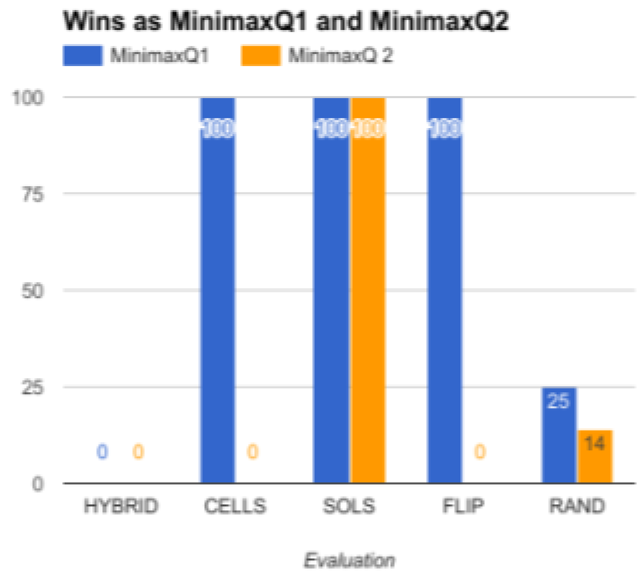

Figure 6: Wins as Minimax agent

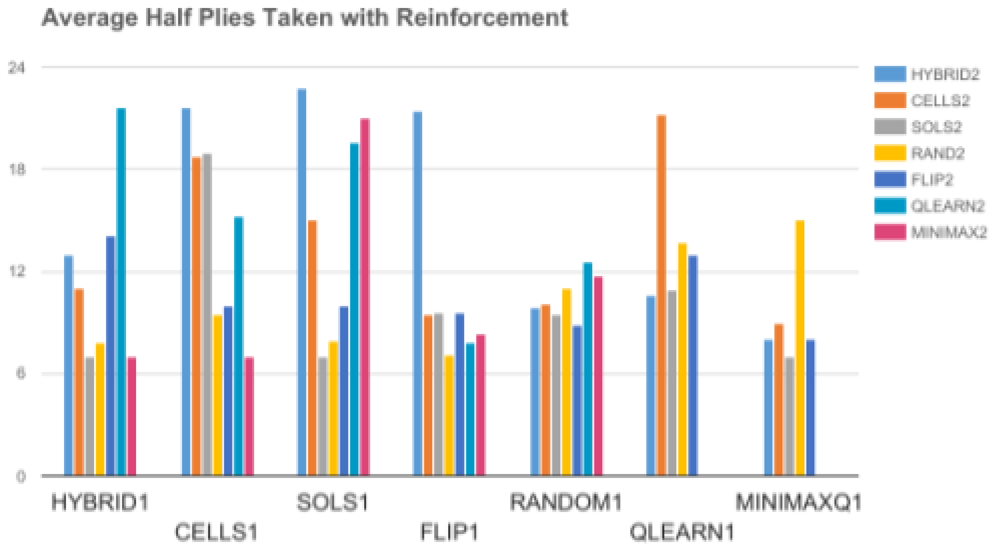

Figure 7: Average Half Plies Taken with Reinforcement

### 4.3 Performance versus Random

Both models underperformed against the random agent in both cases. This is due to random exploiting the amount of possible states by its unexpected behavior. All the other against use a similar path, which allows our reinforcement learning agents to learn the best posibles moves against that particular behavior. This is not possible against the random agents, resulting in poor performances.

## V. Conclusion

In conclusion, we extended a previously build application Conniption to implement some to new reinforcement learning agents, called Qlearn and MinimaxQ. A new simulator class was added, in order to maintain the loop running while the epsilon value is more than the tolerance. We explored with different epsilon decay functions and variables until the optimal was found. Next, all the models were trained (following an automated training scripts) and the dictionary objects were saved for a testing execution. Afterwards, the new created models were evaluated against all the other 5 benchmarks. Both agents performed as expected, with the Qlearn agent failing to generalize a winning solution and Minimax Q getting better results as player 1. Minimax Q was also able to defeat the Flip agent, doing it in a perfect manner, being the only agent in the dataset capable of doing so.

1. **Free-Form Visualization**

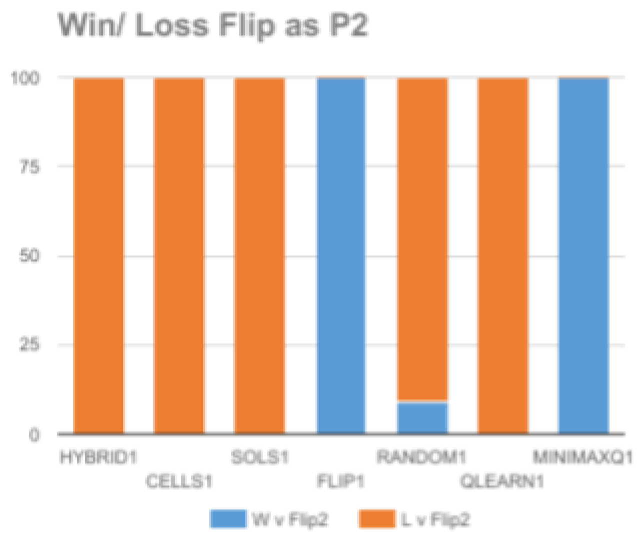

Figure 8: Win/ Loss ratio with Flip as P2

At the conception of the project, the main objective was to be able to implement reinforcement learning agent that would be able to outperform or at least match the previously best algorithm, Flip. In the first iteration of the project Flip was the underwhelming favorite, by dominating both as player 1 and player 2 all the other agents. In figure 8, we can observe how the newly created Minimax Q as player one was the only agent capable of defeating the Flip agent. In such aspect, Minimax Q was able to position itself as a top-2 agent with challenging capabilities against both. Fulfilling in this way the project objective by reaching the desired behavior.

## 2. Reflection

The development of the project was challenging in the sense of being able to add modularity to a well written modular project. Developing the adaptation for the specific cases was challenging, since I encountered some specific edge cases (like the importance of the stage number on a half ply). After the foundations were defined, the finding the accurate variables for the algorithm (epsilon, alpha, epsilon decay) took two weeks of constant experimentation. It was a process which involved a constant debugging, in a matter in which when a new bug was found, all the previous testing was invalid and had to be restarted. It was a long process, but satisfactory when completed.

## 3. Future Work

Reinforcement learning application in Conniption is applicable and possible, however is limited due to the the lack of future prediction embedded in the history of the state's rewards being recorded. The addition of a deep learning network could potentially benefit the created models and give them the edge they required to beat the other minimax agents. In Q Learning, the best possible states are learn through encountering the the same state over different games. However, it does not keeps history of the specific states that will get you there. A deep learning network will in theory add that capability.

The flipping rules, no consecutive flips allowed, create a big benefit to the player executing up to two flips in a half plie and blocking their rival from performing any flips. A slight modification of this rule could improve the fairness of the game.