Team Domination Association

Michael Cline (Project Leader)

Kyle Cibak(Stats/Scapegoat)

Philip Relic (Training Advisor)

Francis Mutuc (PseudoFrank)

April 11, 2011

Contents

Overview

The second reinforcement learning agent was modified from the initial reinforcement learning algorithm using a weighted sum of features for columns and after completing the sum we look ahead 4 moves into a predicted future with a discounting factor of gamma = 0.05. We use features based on states of the board that determine an “optimal move for us to make based on both the current board as well as the predicted future 4 up to 4 moves from now. We decided to design a 2-dimensional vector of feature vectors one for each of the columns. This design choice was discussed in great detail trying to come up with a way to keep possible moves separate while also determining the quality of each of the possible columns to drop in. We trained our algorithm for a long time to try to have our weights converge to the optimal weights for our given feature vector. Initially, similarly to the previous agent the learning program was run against our baseline until it was consistently winning. We then decided to run our new program with look ahead against our previous agent without look ahead. We decided to train against another learning algorithm except this learner didn’t look ahead so presumably this would allow a convergence of our weights that incorporates look ahead. We trained against our previous agent until again consistent winning occurred and our weights were changing minimally after each training epoch. Then, the program was trained against itself so that the opponent will continually be learning and include look ahead along with the agent. In this case, only the first player’s final weights were recorded as the updated weights (because we ran into the problem of both players trying to write to the weights at the same time and our weights getting very messed up), since both players are the same agent, presumably the first player will win the game half of the time. The value for each column was then calculated using the column’s feature vector along with the most current weight vector as well as the discounted sum of the predicted future 4 move sums. Choosing to drop the piece in a specific column was then as simple as choosing the column with the maximum value. Furthermore, if multiple columns shared this maximum value or are in a close enough window to this maximum, a random number from an approximate Normal(5, 25/9) distribution (this was used because we wanted to drop closer to the middle of the board rather than dropping completely in a random column) was obtained and rounded to the nearest integer until an index corresponding to a column with the maximum value is obtained. This manner of randomization in our column choices will presumably prevent repeatedly losing due to deterministically selecting columns.

Data Structure

The main data structures that we used were for representing the board as well as for the feature vectors. For the board, a Vector<Vector<Player>>, which is an 11 x 10 matrix of Player objects, was used where the agent will place himself as ONE in the board, his opponent as TWO in the board, and all other spaces as EMPTY in the board. For the feature vectors (now local to the getOpponentMove function and getMove function), a Vector<Vector<Integer>>, which is an 11 x 17 matrix of Integer objects, was used where the player will place the number of instances of each of the specific 17 features for the first empty row in each of the 11 specific columns. The features that were checked include:

- the agent having a three threat horizontally
- the agent having a three threat vertically
- the agent having a three threat diagonally with an increasing slope
- the agent having a three threat diagonally with a decreasing slope
- the opponent having a three threat horizontally
- the opponent having a three threat vertically
- the opponent having a three threat diagonally with an increasing slope
- the opponent having a three threat diagonally with a decreasing slope
- the agent having a two threat horizontally t
- the agent having a two threat vertically
- the agent having a two threat diagonally with an increasing slope
- the agent having a two threat diagonally with a decreasing slope
- the opponent having a two threat horizontally
- opponent having a two threat vertically
- the opponent having a two threat diagonally with an increasing slope
- the opponent having a two threat diagonally with a decreasing slope
- the row above is a three threat for either the agent or the opponent

Using the getRowFromColumn() function, the first available row for the first column was determined, which was that column's location of interest. Windows were then created, if possible, within 3 spaces in each direction of the location of interest, and the numbers of occurrences of each of the above features were recorded. This process was then repeated for each of the eleven columns. This feature vector was multiplied by the weights in the weights text to come up with a current column sum value. This value is then appended by the discounted sum of the columns predicted from us looking ahead 4 moves. We then play based on the total column sum and with the look ahead we hope that our agent will try to set up traps to force a win.

Weight Adjustment

The weight adjustment process was done adjusting the current weights (stored in a file) by a fraction of their current value (in either direction). We store the history of the game, and then after we have a result, we walk through the history, starting from the last move, and

$$\forall\, i, m \in weights, weights[i] = weights[i] + \frac{\lambda^{n-m} * r * feat[i] * |weights[i]|}{\beta}$$

$where:$
$\boldsymbol{weights}[\boldsymbol{i}]$ $is\ the\ current\ weight\ for\ the\ i^{th}\ feature$
$\boldsymbol{\lambda}$ $is\ the\ learning\ rate$
$\boldsymbol{n}$ $is\ the\ total\ number\ of\ moves\ of\ the\ completed\ game$
$\boldsymbol{m}$ $is\ the\ move\ in\ the\ history\ we\ are\ examining\ (n-m=0)$
$\boldsymbol{r}$ $is\ the\ result\ of\ the\ game\ \{-1,1\}$
$\boldsymbol{feat}[\boldsymbol{i}]$ $is\ the\ value\ of\ the\ i^{th}\ feature$
$\boldsymbol{weights}[\boldsymbol{i}]$ $is\ the\ current\ value\ of\ the\ current\ weight$
$\boldsymbol{\beta}$ $is\ a\ regularization\ factor$

To determine whether to alter the weight, an indicator function was used to show which features were present for the chosen column of each move. Only the weights that were non-zero were altered.

The first 4 features (our win-conditions), are divergent features. These features can never go down, because we utilizing one of these 'win' features never leads to a lost. Thus, we placed a cap of 10 on these values in order to avoid having these weights diverge. Also, there was a concern that the negative weight representing 3-in-a-row in the row above to outweigh the positive weight representing the features for the 3-threats in the current row. In order to solve this problem, the absolute value for the weight representing the features for the 3-threat in the row above was capped to be .1 less than the 3-threat so that the 3-threat will still be chosen most of the time even if it is also a column where the row above the first empty row also has a 3-threat.

Training

Initially, in order to train the weights, the learning agent was trained against the baseline for one epoch where one epoch was 50 consecutive games. The learning agent beat the baseline consistently, with only a few losses due to the baseline obtaining “luckier” random starts than the learning agent. After that, we trained our learner against our previous agent for 10 epochs. In order to avoid over fitting the weights, the learning agent was then trained against itself, an opponent who was also able to learn and adapt its strategy. This process of running the learning agent against itself was done for 50 epochs. The decision to stop here was made as we saw small oscillations in the range of numbers that continued to get smaller and smaller giving us the indication of semi-convergence.

Improvements From the Previous Agent

Trials and Tribulations of Look-Ahead: As an improvement from the previous baseline, we decided to implement look-ahead and an exploration strategy. We decided to implement a recursive look-ahead where we simulated an opponent's move by giving them their own feature vector, from which they use to make their 'optimal' move. To do this, we had to readjust our implementation from our previous baseline. Previously, our learner class had its board, feature vector, and other statistics as global members. When we decided that we needed to maintain these statistics for our opponent as well, we had to deal with collision between the opponent's board and feature vector, and our board and feature vector. To do this, we maintain only one global statistic, which is the current game board. We changed most of our board manipulation, feature generation, and statistic manipulation functions to take a board as a parameter. By doing this, we are able to pass in the global board or boards with hypothetical moves that we are thinking about taking, or simulating our opponents taking. There were a few issues that we ran into when implanting the look-ahead. First, as we look farther into the future, we generate an exponential number of hypothetical boards, and given time constraints, limit the depth with which we are able to recurse. Another problem we ran into was getting the discounting factor correct. When our factor was too high (gamma of .1), we ran into problems where we were not blocking threats because we saw that we could win in the future. When we set our gamma factor too low (.01), then the strategy gained from looking ahead became negligible. By setting our discounting (gamma) factor to .05, we found a balance where our learner would never let an opponent win just because it thought it could win in the future, and it would also cause a difference when debating between similar moves.

Into the Unknown: Developing an Exploration Strategy: Our previous learner also did not implement an exploration strategy, and as a result, was completely predictable, and our decisions were completely based on our weights during that current game. We implemented the epsilon-greedy strategy, where every time it was our turn to move, we took a random move with a small probability, and otherwise used our feature vector to pick an optimal move. In addition, because exploration frequently led to a quick loss, we restricted the ability to explore to when we were training. By doing this, we are free to explore and find better solutions (weights) when we are training, but when we are in play-mode, we aren't exposed to the risk of exploring a solution that will lead to a quick loss. We also varied our epsilon value (using 50%, 25%, 10%) to add noise to our weights. We then moved to an epsilon value of .01 (1%) so our weights would settle.

Weaknesses

Our last agent's biggest weakness was its inability to look ahead. We have modified our second agent to include a look-ahead. It's a depth-recursive look-ahead where we simulate an opponent's move. Currently, we look-ahead 4 moves to determine whether future moves will impact our current move. However, this foresight is still somewhat limited. Even when we greatly discount future observations, it is still possible that many less important features will have a greater weight than one important feature, and thus we will choose a bad move based on future guesses.

Our previous agent did not implement any exploration strategy. We have implemented an epsilon-greedy exploration strategy where every 1%, we choose a random legal move. A possible weakness of this strategy is that when we explore, we do not take into account the quality of the random move taken. To improve upon this, a better exploration strategy, such as the soft-max exploration strategy, should be utilized.

Strengths

We found that semi-quantizing our win-condition features and our ‘do-not-place’ feature led to much better performance. We capped the win-condition features at 10 so they would not diverge. We placed a variable cap on our ‘do-not-place’ feature, which is the minimum of any feature that will lead to a game-ending play. By doing this, we ensured that the ‘do-not-place’ value never over-rides taking a win or blocking the opponent’s ability to win.

There a number of features that we added to our previous agent. The first is the look-ahead feature. We use a recursive look-ahead to simulate an opponent’s move based on giving them a simulated feature vector, and letting them make an ‘optimal’ move. After they make their move, we can respond by adjusting our feature vector accordingly. We also discount future guesses by a polynomial variable factor, because current board positions are more important than hypothetical future ones.

Another strength is our exploration strategy. Without an exploration strategy, all of our moves will be determined solely by our weights, and as a result, will be stuck without a chance to find a more optimal strategy. We have implemented an epsilon-greedy strategy where some percent of the time, we will make a random move. This way, we give ourselves the chance to randomly wander into a situation that leads to a favorable outcome, in which case our weights will be slightly adjusted to reflect that, depending on how far back in the history that random move affected our outcome.

Also, we changed our random move to be slightly more strategic. Instead of choosing a completely arbitrary move, we choose a move that is within a certain distance from the maximum sum. So instead of choosing a random move with a very low sum, we choose a move that isn’t the maximum sum, but is within a certain amount of the maximum sum. In this way, even our random moves have an element of strategy to them, and we can take random moves that will tend to be more helpful.

Algorithms

```
setupBoard()            // creates a copy of the game board and stores it in a global variable
```

```
initWeights()           // initializes the weights of each feature vector to 0
  1 for i = 1 to numFeatures
  2   Weight of F[i] = 0
```

```
threeHorizontal(Board, Player, col, row)
  1   Check all available horizontal windows of four spaces for all possible combinations of
      location row, col in current board
  2   If a space and three of Player’s our pieces is found in any of these windows, then return
      found.  Otherwise, return not found
```

```
threeVertical(Board, Player, col, row)
  1   if row = 3
  2     return not found
  3   for each possible combination of creating three in a row vertically in col and row in the
      current state of board
  4     Check all available vertical windows of four spaces for each column in board
  5     If a space and three of Player’s pieces is found in any of these windows, then return
```

```
        found.  Otherwise, return not found
```

threeDiagRight(Board, Player, col, row)

```
1   for each possible combination of creating three in a row diagonally to the right in col,
    row in the current state of board
2     Check if there exist three of Player's pieces in col and row
3     If three of a Player's piece is found, return found.  Otherwise, return not found
```

threeDiagLeft(Board, Player, col, row)

```
1   for each possible combination of creating three in a row diagonally to the left in col,
    row in the current state of board
2     Check if there exist three of Player's pieces in col and row
3     If three of a Player's piece is found, return found.  Otherwise, return not found
```

twoHorizontal(Board, Player, col, row)

```
1   for each possible combination of placing two of Player's pieces in a window spanning
    four horizontal spaces up starting from col to row in the current state of board
2     Check if there exist two of Player's pieces within the window
3     If two of a Player's piece is found, return found.  Otherwise, return not found
```

```
twoVertical(Board, Player, col, row)
  1   for each possible combination of placing two of Player's pieces in a window spanning
      four vertical spaces up to row in the current state of board
  2       Check if there exist two of Player's pieces within the window
  3       If two of a Player's piece is found, return found.  Otherwise, return not found
```

```
twoDiagRight(Board, Player, col, row)
  1   for each possible combination of placing two of Player's pieces in a window spanning
      four spaces diagonally to the right up starting in col and going up to row in the current
      state of board
  2       Check if there exist two of Player's pieces within the window
  3       If two of a Player's piece is found, return found.  Otherwise, return not found
```

```
twoDiagLeft(Board, Player, col, row)
  1   for each possible combination of placing two of Player's pieces in a window spanning
      four spaces diagonally to the left up to row in the current board
  2       Check if there exist two of Player's pieces within the window
  3       If two of a Player's piece is found, return found.  Otherwise, return not found
```

```
doNotPlace(Board, col, row)  // determines if a col should be taken away as a possible
                             // move at for a given state of the board
1 for i = 0 to 8
2     Check if there exists a threat of each kind for each Player in col and row+1
3     If a threat is found, then put that column in the feature vector.  Otherwise, put column 0
      in the feature vector.
```

```
weightUpdate()    // iterate through feature vectors log and update the weights based on the
                  // results of the game played; our learning rate is raised to n-m, where n is the
                  // total number of moves in a game, m is current move of the game because we
                  // want to have our weights change more for moves later in the game; our
                  // indicator function is a flag that denotes if a feature is present in the current
                  // vector; we use our indicator function to normalize our adjustments of our
                  // weighs because we found that our adjustments sometimes caused certain
                  // weights pivotal to us winning the game to become negative and other times
                  // we ran into the issue of our weights canceling out to zero and causing
                  // problems for future updates

1 for i = 0 to numFeatures
2     Weight for feature i += learning rate$^{n-m}$ * indicator function * |weight for feature i| / 100
```

```
readWeights()        // reads in the weights from our weights textfile
                     // no voodoo magic in here; it is as simple as it sounds
```

```
getMove(Board)          // recursively looks ahead 4 of our moves from the  current state of
                        // the board by simulating our predicted moves and our opponent's
                        // predicted moves; at each level of our recursion we initialize and
                        // set our feature vectors depending on the current state of the
                        // board; then for that specific recursive step we get the sums of each
                          // column and make a move in the column with the largest sum as
                          // ourselves; If there are multiple columns with a sum within .01 of
                          // the max sum of these 11 columns, then we randomly choose
                          // between these columns; in the final phase of our recursive step
                          // we make a move as the opponent calling by calling the
                          // getOpponentMove function.  We also implement bounds
                          // checking for looking ahead to insure that we are not looking
                          // ahead near the end of the game because the board would be close
                          // to full and there would be no benefit in looking ahead 3 moves
                          // when a board scenario as only 1 more possible move.

for i = 1 to 4
   for j = 0 to numberOfCols
      for k = 0 to numFeatures
         Sum += F[j] * Weight[j]
          if Sum < 1
          DropOurPiece(colSumsArray, Sum) // random move
          return maxCol
```

```
9    getOpponentMove(Board)


getOpponentMove(Board)            // simulates an opponent's move; at the start we initialize
                                  // and set the opponent's feature vectors depending on the
                                  // current state of the board; then we get the sums of each
                                  // column and make a move as the opponent in the column
                                  // with the largest sum as ourselves; If there are multiple
                                  // columns with a sum within .01 of the max sum of these
                                  // 11 columns, then we randomly choose between these
                                  // columns.
1    for i = 0 to numberOfCols
2        for j = 0 to numFeatures
3            Sum += F[i] * Weight[i]
4            if Sum < 1
5    DropOurPiece(colSumsArray, Sum)   // random move
6    return maxCol


DropOurPiece(Board, colSumsArray, max)   // drops piece in a random column that has a
                                         // column sum close to the max column in the
                                         // current state of board
```