Learning to play Nim with Reinforcement Q-Learning

Karl Kintner-Meyer

June 1, 2020

Contents

Abstract

Creating a program that can learn to successfully play games at a human or super-human level is a long-standing goal for machine learning research and a significant stepping stone in using machine learning to determine optimal policies in highly complex techno-economic situations. Different machine learning algorithms have found various levels of success in games with different complexities, from simple games like Tic-Tac-Toe, to highly complex games like Chess and Go. Part of the challenge of applying machine learning to games is determining which algorithm will perform best for a certain game complexity level. In this report, we apply a Q-learning algorithm to train an AI to play Nim, a game with moderate strategic complexity. Using a matrix that stores payoffs for every state-move pair, we allowed an AI to play against itself using the payoff matrix in its decision-making process. After training, we show that the AI successfully learned all the optimal moves that result in a forced win. The AI was successfully able to win any game from a guaranteed-win state, as well as correctly punish sub-optimal opponent moves that left guaranteed-win states open to the AI. The work done in this report shows that a Q-learning approach can be an effective strategy for training an AI to play games of moderate strategic complexity.

1 Background on Reinforcement Q-Learning

At its core, reinforcement learning is very similar to human learning. The computer learns a policy by interacting with an environment and recieving occasional feedback, either positive or negative. The computer then stores the cumulative feedback in memory and uses this feedback to inform its future decisions. In the case of Q-learning, ”memory” takes the form of a Q-table, a matrix that has rows equal to the number of possible states and columns equal to the number of actions that can be taken. Thus, every cell in the matrix denotes a state-action pair, with the cell being updated based on feedback from computer interaction with the environment. With a large enough sample size and a proper balance of exploration of the state space and exploitation of well-rewarded actions, the computer can discover optimal policies.

One of the main properties of Q-learning is that it is model independent. In some sense, this is a significant advantage as modelling error can often cause significant performance degredation in machine learning algorithms, especially in applications that are highly sensitive to parameters. Q-learning circumvents this risk by creating a straight line association between actions and their respective ”values” and avoiding an environment model altogether. Figure 1 illustrates one iteration through the Q-learning process. An action sequence is input to the environment, and is either positively or negatively reinforced. Thus, through successive iterations through the Q-learning process, individual actions in the action sequence can be altered to generate consistent positive reinforcement.

Input Action Sequence
Environment
Reinforced Response

Figure 1: Illustration of Q-learning Process

Q-learning's model independence also has a few noteworthy tradeoffs. Firstly, Q-learning can have significant interpretability issues. Referring back to Figure 1, we note that the algorithm never attempts to draw conclusions about the "environment" black box. This approach yields very little insight into why the given action sequences are positively or negatively reinforced and does not readily allow for general conclusions about the environment to be drawn from optimal action sequences. Furthermore, Q-learning usually is applied on a very case specific basis. Consider the case in which the environment changes slightly from one training session to the next. Since the Q-learning algorithm drew no conclusions about the environment pre-change, there is no guarantee that the optimal action sequences pre-change will be anywhere near optimal post-change.

A second notable tradeoff in Q-learning is the high sample size requirement and lack of guaranteed convergence. Q-learning relies heavily on the ability to collect a large enough sample size of action sequences such that optimal individual actions can be discovered through successive reinforcement trials. As such, training on any kind of environment with a large state space or action space requires an extremely large number of samples to extract the optimal policy. Furthermore, there is no guarantee that the training algorithm will converge to an optimal policy in a reasonable amount of time. For some environments with large state or action spaces, the algorithm may stay far away from the optimal policy and fail to ever find a reasonably successful action sequence.

2 Background on Nim

Nim is a two-player game of sticks and stacks. In general, a Nim board contains n stacks, with each stack containing s_i sticks. The rule of the game is simple: on a player's turn, they must remove any positive number of sticks from the Nim board with the constraint that all sticks must be removed from a single stack. The goal of the game is to force the opponent to remove the last remaining stick on the Nim board. For this project, we focus on the Nim board with $n = 4$ and stack sizes: $s_1 = 2, s_2 = 3, s_3 = 4, s_4 = 5$, which can be conveniently represented by a column vector: $[2, 3, 4, 5]^T$. It is also known a-priori that there are several game substates that can guarantee victory for a player if he makes a move that leaves his opponent in the substate. The relevant game-winning substates for this Nim board are: $[n, n]^T$, $[m, m, n, n]^T$, $[1, 1, n, n]^T$, and $[1, x, x + 1]^T$ with x odd. A brief explanation of optimal move sequences for each substate is as follows:

$[n, n]^T$ substate:
The goal for the player who leaves his opponent in the $[n, n]^T$ substate is to continue reducing the game board down to successively smaller $[n, n]^T$ states until there is an opportunity to remove all sticks remaining on the board except one. As an example, we will consider the $[2, 2]^T$ substate. An opponent who is left with the $[2, 2]^T$ substate must remove either 1 stick or 2 sticks from one of the stacks. If he removes one stick from a stack, the winning player removes both sticks from the other stack, leaving one stick remaining on the board. Alternatively, if the opponent removes 2 sticks from one stack, the winning player removes one stick from the other stack, also leaving one stick remaining on the board. Thus a player who leaves his opponent with an $[n, n]^T$ state can always successively reduce to smaller $[n, n]^T$ substates or win the game outright. An additional property of note with this substate is that a player who leaves his opponent in an $[n, n]^T$ substate can also force a loss for himself. This can be shown by using the same successive reduction of $[n, n]^T$ states but playing to force himself to take the final stick. Thus, a player who leaves his opponent in an $[n, n]^T$ substate has total control over the

outcome of the game. This property will be useful for proving larger guaranteed-win substates.

$[1,1,n,n]^T$ substate:
The goal for the player who leaves his opponent in the $[1,1,n,n]^T$ substate is to force the game into an $[n,n]^T$ substate or continue playing just the $[n,n]^T$ part of the board while ignoring the $[1,1]^T$ part of the board. If an $[n,n]^T$ substate is reached, it is a guaranteed win. If the winning player plays only the $[n,n]^T$ part of the board, the opponent will be forced to remove the last stick of the $[n,n]^T$ part of the board, which, by the rules of the game, guarantee that the opponent will also remove the last remaining stick on the overall board.

$[m,m,n,n]^T$ substate:
The goal for the player who leaves his opponent in the $[m,m,n,n]^T$ substate is to force the game into a $[n,n]^T$ substate. Here we use the property of the $[n,n]^T$ substate having total control over the outcome of the game and consider the $[m,m]^T$ section of the board separately from the $[n,n]^T$ section. The player who leaves his opponent in this substate will want to force a "loss" in either the $[m,m]^T$ section or the $[n,n]^T$ section which guarantees that they will draw the last stick in either of the sections, reducing the game to a single $[n,n]^T$ substate which is a guranteed win.

$[1,x,x+1]^T$, x odd substate:
The goal for the player who leaves his opponent in this substate is to further reduce this state to successively smaller $[1,x,x+1]^T$, x odd substates, to an $[n,n]^T$ substate, or to a $[1,1,1]^T$ substate.

With these optimal moves in mind, it can be shown that with perfect play, player 2 should always be able to win on the $[2,3,4,5]^T$ Nim board.

3 Methods

The creation of a trainable AI required two separate routines. One routine to simulate the game and enforce the rules, and another routine to act as the AI. To train the AI to play Nim, we built a "state-move-pair payoff table" also referred to as a "Q-table" that contains rows equal to the number of possible game states and columns equal to the number of possible moves. The game simulation routine was programmed to sort the rows of the game board after every move to reduce the total number of possible game states (i.e. $[0,2,2,0]^T$ and $[0,0,2,2]^T$ are equivalent game states).

In order to balance the exploration of the state space with the exploitation of discovered optimal moves, we also included another parameter in our Q-learning algorithm, ϵ, which is defined to be the percentage of time the computer picks a random move. In any case where a random move is not picked, the computer picks the highest scored move for the given state. For our purpose, a decaying ϵ strategy was used. In the early stages of training, we used a high ϵ value to facilitate high exploration of the state space, then gradually decreased it as the AI went through more and more training cycles in order to take advantage of identified optimal moves.

Both routines were implemented in MATLAB. The game routine steps through successive turns of the game and, depending on what mode was selected, requests a move from either a human player or the AI player. The AI routine is responsible for returning a move given the input game state and value of ϵ. The AI routine uses the Q-table to make its decision, however, the simulation routine is responsible for updating the Q-table after each game. A schematic of the two routines working in tandem is shown in Figure 2 below.

The computer was allowed to play against itself, saving a per-game list of all of its moves as player 1 and player 2. After each game, the Q-table was updated using the list of moves from both players, incrementing the score of every state-move pair in the Q-table that appears in the winners list and decrementing the score of every state-move pair in the Q-table that appears in the loser's list.

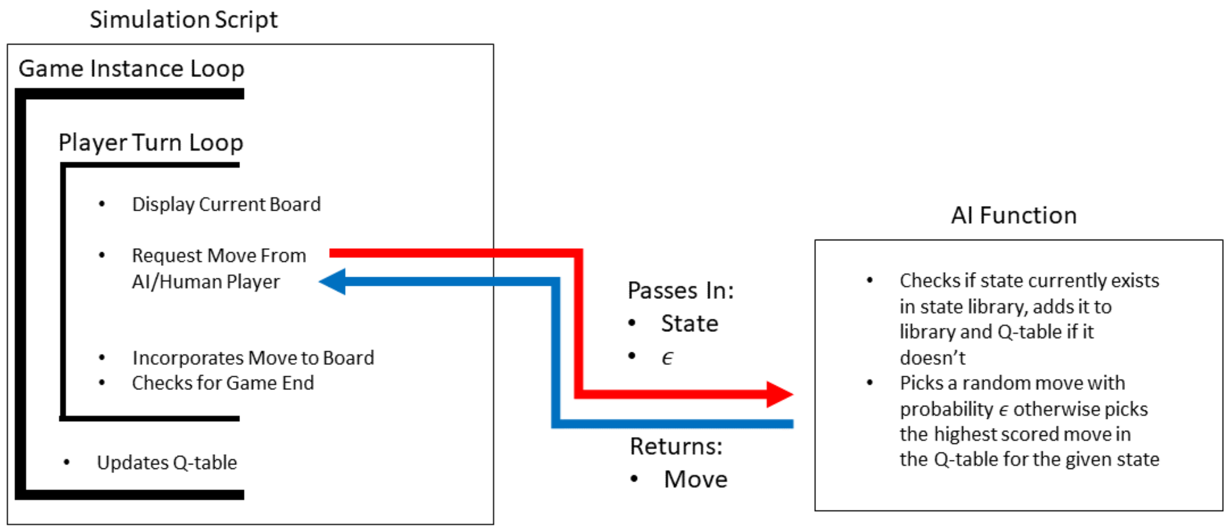
Simulation Script
Game Instance Loop
Player Turn Loop
Display Current Board
Request Move From AI/Human Player
Incorporates Move to Board
Checks for Game End
Updates Q-table
Passes In:
State
ϵ
Returns:
Move
AI Function
Checks if state currently exists in state library, adds it to library and Q-table if it doesn't
Picks a random move with probability ϵ otherwise picks the highest scored move in the Q-table for the given state

Figure 2: Algorithm Implementation

4 Results and Discussion

The AI was trained on the following training schedule: 12,000 games with an ϵ value of 0.99, 9,000 games with an ϵ value of 0.7, 6,000 games with an ϵ value of 0.5, and 3,000 games with an ϵ value of 0.3. After training, it was determined that every state in the Q-table with a winning move had the corresponding move correctly identified by the AI.

During training, there were several surprising patterns of note that are worth discussing. The first pattern was the AI's struggle to find good move sequences in the early stages of training in the high-ϵ regime. The Q-table update was implemented in such a way that sequences of moves would either be all rewarded or all punished. In doing this, the AI could only discover good individual moves via a large enough sample of move sequences that contained the good move. In the early stages of training, most moves were picked at random and thus, a good early-game move could easily be cancelled out by a poor late-game move, resulting in both the good move and bad move being punished. Likewise, the computer could make a poor early-game move, but follow up with a good late-game move and both the poor move and good move would be rewarded. This relationship between early-game moves and late-game moves made it very difficult for the AI to accurately determine the quality of moves in earlier game states and required a much larger number of move sequence samples.

In contrast, for game states that were close to the end of the game, the AI was much faster at determining optimal moves. Notably, it was quick to discover the game winning moves for the $[0,0,2,2]^T$ and $[0,1,2,3]^T$ substates. Since these moves were much closer to the game's end, the search space was much smaller in comparison to the early game substates, and thus, the AI was able to more quickly converge to optimal moves for these states. In addition, once the training schedule reached the medium-ϵ regime, the AI was much faster at finding optimal early-game moves. Since the AI is more focused on exploiting known optimal moves at lower ϵ values, the AI only had to search the early-game move tree until it found a previously known game-winning state, at which point it would play itself to victory. In this way, the AI learned using a bottom-up learning approach, first using random sampling to determine winning states near the game end, and then using exploitation to search forward through the early-game states to find known game-winning substates.

The second notable pattern that emerged during training was the AI failing to properly reinforce itself as its own opponent. In the early stages of training, where the AI had very little information about quality of moves, it failed to punish bad moves as its own opponent, and as a result, several poor moves were incorrectly rewarded and several good moves were incorrectly punished. This behavior continued somewhat until the training entered the medium-ϵ and low-ϵ regimes, at which point the AI was punished repeatedly for drawing incorrect conclusions about move quality in the early stages.

5 Future Work

With the knowledge gained from this project, an important future step to take would be to test how well this algorithm scales up to a larger Nim board. Namely, one with hundreds of stacks and hundreds of sticks per stack. In a future trial, we would expect that it would be much more difficult for an AI to recognize good early-game moves due to the sheer size of the move tree in larger game boards. An alternative approach could be to use a separate machine learning algorithm as a Q-value function approximator to estimate what the Q-values should be for each state-move pair, and train both the approximator and the Q-learning algorithm in tandem.

6 Appendix A

The following pages contain sample games between the trained AI and the Author. Note that the notation $[rowNum, valueNum]$ indicates the row number, and the number of sticks to be removed from the row. Game board rows are sorted after each move.

```
AI As Player 2, Perfect Play.

Player 1's Turn

Current Game State:

A =

     2
     3
     4
     5

Please input value in following notation, [rowNum,valueNum][1,1]
Player 2's Turn

Current Game State:

A =

     1
     3
     4
     5

Player 1's Turn

Current Game State:

A =

     0
     1
     4
     5

Please input value in following notation, [rowNum,valueNum][2,1]
Player 2's Turn

Current Game State:

A =

     0
     0
     4
     5

Player 1's Turn
```

```
Current Game State:

A =

     0
     0
     4
     4

Please input value in following notation, [rowNum,valueNum][3,1]
Player 2's Turn

Current Game State:

A =

     0
     0
     3
     4

Player 1's Turn

Current Game State:

A =

     0
     0
     3
     3

Please input value in following notation, [rowNum,valueNum][3,1]
Player 2's Turn

Current Game State:

A =

     0
     0
     2
     3

Player 1's Turn

Current Game State:

A =
     0
     0
     2
     2

Please input value in following notation, [rowNum,valueNum][3,1]
Player 2's Turn

Current Game State:

A =

     0
     0
     1
     2

Player 1's Turn

Current Game State:

A =

     0
     0
     0
     1

Please input value in following notation, [rowNum,valueNum][4,1]
Player 1 Loses
```

```
AI As Player 1, exploiting Player 2 error on P2-move 1

Player 1's Turn

Current Game State:

A =

     2
     3
     4
     5

Player 2's Turn

Current Game State:

A =

     0
     2
     4
     5

Please input value in following notation, [rowNum,valueNum][3,3]
Player 1's Turn

Current Game State:

A =

     0
     1
     2
     5

Player 2's Turn

Current Game State:

A =

     0
     1
     2
     3

Please input value in following notation, [rowNum,valueNum][3,2]
Player 1's Turn
```

```
Current Game State:

A =

     0
     0
     1
     3

Player 2's Turn

Current Game State:

A =

     0
     0
     0
     1

Please input value in following notation, [rowNum,valueNum][4,1]
Player 2 Loses
```

7 Appendix B

The following pages contain the MATLAB code used to train the AI.

```
clear all; close all; clc;
gameMode = 1; % 0 for AI training, 1 for AI as P1, 2 for AI as P2, 3 for 2 player/no ↙
AI
epsilon = 0; % factor for how much exploring is done, only should be used for ↙
training
N = 0; % will be used to create a for loop around the whole game sim, only one ↙
iteration if playing against AI, but N iterations for training the AI.
if gameMode == 0
    epsilon = 0.1;
    N = 100;
else
    epsilon = 0.001;
    N = 1;
end

P2wins = 0;

for iter = 1:N
GameStates = zeros(4,1);
MoveList = zeros(1,2);
A = [2;3;4;5];
turn = 1;
while sum(A) ~= 0
    turn = turn + 1; % increment turn count

    % store values of game state -----------------
    GameStates = [GameStates,A];

    % display useful information ------------------
    currentPlayerNum = mod(turn,2) + 1;
    disp("Player " + string(currentPlayerNum) + "'s Turn");
    disp(" ");
    disp("Current Game State:");
    A

    % initialize values to get inside the while loop ---
    x = [1,10];
    rowNum = x(1);
    valueNum = x(2);

    gaveInvalidMoveFlag = -1;
    SearchedRow = 0;
    while A(rowNum) < valueNum
    % requesting valid input from user or AI --------------
    gaveInvalidMoveFlag = gaveInvalidMoveFlag + 1;
    if gameMode == 0
        [r,x] = NimAI(A,epsilon, gaveInvalidMoveFlag, SearchedRow);
        SearchedRow = r;
    elseif gameMode == 1 && currentPlayerNum == 1
```

```
        [r,x] = NimAI(A,epsilon, gaveInvalidMoveFlag, SearchedRow);
        SearchedRow = r;
    elseif gameMode == 2 && currentPlayerNum == 2
        [r,x] = NimAI(A,epsilon, gaveInvalidMoveFlag, SearchedRow);
        SearchedRow = r;
    else
        x = input("Please input value in following notation, [rowNum,valueNum]");
    end

    rowNum = x(1);
    valueNum = x(2);
    end


    % attempt to perform operation on nim board -----------------
    A(rowNum) = A(rowNum) - valueNum;
    B = sort(A);
    A = B;



    % store move in movelist ----------------------------
    MoveList = [MoveList;x];


end

disp("Player " + currentPlayerNum + " Loses");

if currentPlayerNum == 1
    winningPlayerNum = 2;
    P2wins = P2wins + 1;
else
    winningPlayerNum = 1;
end

% update Qtable from move list and state list ---------
load('Qtable.mat','StateLookup','MoveLookup','Q_Table');
GameStates = GameStates(:,2:end);
MoveList = MoveList(2:end,:);
% Reward good moves
for k = winningPlayerNum:2:size(GameStates,2)
    z_state = GameStates(:,k);
    z_move = MoveList(k,:);
    for j = 1:size(StateLookup,2) % Search across all columns of the state lookup↙
table to see if state is in table already
        if sum((z_state == StateLookup(:,j)) - ones(4,1)) == 0 % if the state is in↙
the table
            break; % break out of loop, note that the last j value before the break↙
```

```
represents the column of the state in the state lookup table
        end
    end
    Q_Table_rownum = j;
    for j = 1:size(MoveLookup,1)
        if sum((z_move == MoveLookup(j,:)) - ones(1,2)) == 0
            break;
        end
    end
    Q_Table_colnum = j;
    Q_Table(Q_Table_rownum,Q_Table_colnum) = Q_Table(Q_Table_rownum,Q_Table_colnum) + ↙
1;
end


% Penalize Bad Moves
for k = currentPlayerNum:2:size(GameStates,2)
    z_state = GameStates(:,k);
    z_move = MoveList(k,:);
    for j = 1:size(StateLookup,2) % Search across all columns of the state lookup↙
table to see if state is in table already
        if sum((z_state == StateLookup(:,j)) - ones(4,1)) == 0 % if the state is in↙
the table
            break; % break out of loop, note that the last j value before the break↙
represents the column of the state in the state lookup table
        end
    end
    Q_Table_rownum = j;
    for j = 1:size(MoveLookup,1)
        if sum((z_move == MoveLookup(j,:)) - ones(1,2)) == 0
            break;
        end
    end
    Q_Table_colnum = j;
    Q_Table(Q_Table_rownum,Q_Table_colnum) = Q_Table(Q_Table_rownum,Q_Table_colnum) - ↙
1;
end

save('Qtable.mat','StateLookup','MoveLookup','Q_Table');


end


%P2wins
```

```
function [StateRow,move] = NimAI(state,epsilon, gaveInvalidMoveFlag, prevJ)
    load('Qtable.mat','StateLookup','MoveLookup','Q_Table'); % load Q-table and↙
lookup tables
    stateInTableFlag = 0; % set state in table flag to false;

    if gaveInvalidMoveFlag == 0 % if an invalid move has not been given yet
    % Check first if the state exists in the StateLookup table ----------
    for j = 1:size(StateLookup,2) % Search across all columns of the state lookup↙
table to see if state is in table already
        if sum((state == StateLookup(:,j)) - ones(4,1)) == 0 % if the state is in the↙
table
            stateInTableFlag = 1; % make state in table flag true
            break; % break out of loop, note that the last j value before the break↙
represents the column of the state in the state lookup table
        end
    end

    if stateInTableFlag == 0 % if the state in table flag remains false
       j = j+1; % increment j to account for new column
       StateLookup = [StateLookup,state]; % append the newly discovered state onto↙
the state lookup table
       Q_Table = [Q_Table; zeros(1,size(Q_Table,2))]; % add a new row in Qtable for↙
newly discovered state
    end

    else
        j = prevJ; % if invalid move has already been given, just use previous j↙
value
    end

    % Decide on a move to make ------------------------------
    x = rand; % generate random variable from uniform distribution to determine↙
whether to exploit or explore
    if x < epsilon % if less than epsilon value, explore. Thus epsilon is the↙
percentage of how often exploring is done
        y = randi(size(MoveLookup,1)); % generate random integer in the range of 1-↙
number of possible moves
    else % if greater than epsilon, exploit.
        [value,y] = max(Q_Table(j,:)); % search for highest value in Q table row↙
corresponding to state
    end

    move = MoveLookup(y,:); % return move
    StateRow = j;

    save('Qtable.mat','StateLookup','MoveLookup','Q_Table'); % save variables back to↙
mat file.

end
```