MAY 1, 2019

ENPM 808F

PROJECT 4: REINFORCEMENT LEARNING

TIMOTHY WERDER

Table of Contents

0
Abstract 2
Introduction 2
Dots and Boxes 3
Approach 4
Q Learner 5
Functional Approximation 5
Results 6
Q-Learner 6
Functional Approximator 8
Benchmarking 9
Analysis and Future Work 10
Drop in Performance 10
100 Game Discrepancy 10
Improving Overall Performance 10
Future work 10
References 12
Code 13
Matlab 13
Main 13
RLearner 25
Greedy 25
QPolicy 26
QPolicy2 27
Random 29
isBox 29
getSate 29
writeoutDL 30
Python 30
DLQApprox 30
DLQApprox3x3 32
DLQOut 33

Abstract

Reinforcement Q Learning with Dots and Boxes

Timothy Werder, University of Maryland, College Park

Reinforcement Learning is a fairly new field of study having only been around for 30 years. It has been developed on since its inception in the 1980s and has mostly been used to demonstrate a machines ability to learn tasks and strategies, namely respective ones with fixed possible actions and states. In this project, the task was to teach a reinforcement agent to learn how to play Dots and Boxes, a game where two or more players take turns drawing lines between dots to create boxes. The player that completes the greatest number of boxes wins the game. This was to be done using two forms of Reinforcement Learning known as Q Learning and Functional Approximation.

The findings of this project show that for smaller games (2x2) that there is a cutoff for the number of training episodes that an agent can learn before suffering from overtraining. This number appears to be related to the number of possible states in that game, 4096. This is true for both the Q Learner and the Functional Approximator. However, prior to over training either could win 80-90% of games against a player making random moves. For the 3x3 game, which has a possible 16 million plus possible states, the win percentage showed improvement as the number of training episodes increased. Additionally, the Functional Approximator for the 3x3 had the highest number of wins due to the fact that Functional Approximators rely more heavily on the action rather than the state to determine the next move. The Q Learner capping out at 60-70% win rate and the Functional Approximator winning 90+% of all games.

Introduction

Reinforcement Learning has been an up and coming area of research and use in the field of Ai. In particular Reinforcement learning has been used in many different games to test the ability of humans in bot their their ability to play and their ability to create machines that are designed and trained to win. One form of Reinforcement learning is Q Learning, a branch that uses an iterative policy development, evaluation, and reward model to train an agent to play a game or accomplish a task.

In this project, the goal was to build a Q Learner and a Functional Approximator of a Q Learner to play the game of Dots and Boxes. The machine (red) would then play against itself (blue) to learn how to win. The machine would be rewarded 1 point for securing a box and 5 points for winning a game. After training the machine would then play against a random bot (pink) to test how well it had learned the game. The machine was then evaluated based on how many games it won against the random bot out of 100 games.

Next the Functional Approximator (FA) would receive the state and action pairs from the Q Learner's training and would build a model to win the game. It would then play against the random opponent and eb evaluated in the same structure that the Q Learner was evaluated.

Dots and Boxes

The game of dots and boxes is a relatively simple game at face value. Two, or more, players will draw up a series of dots in rows and columns.

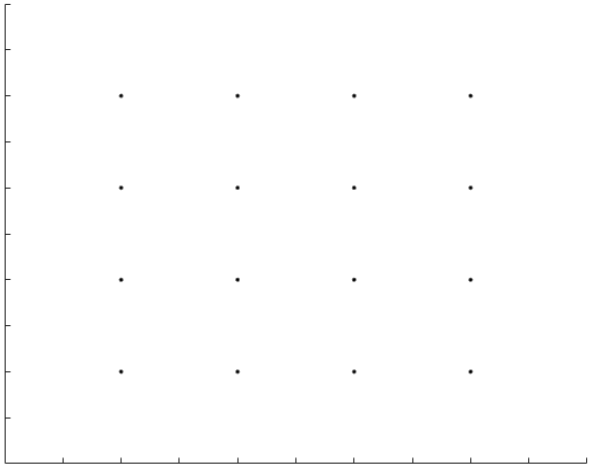

Figure 1: Dots and Boxes game Dot Initialization 3x3 game

The players will then take turns drawing a line until one player creates a box. That player will then collect a "point" for completing the box and will be allowed to take another turn, repeating until that player does not create another box.

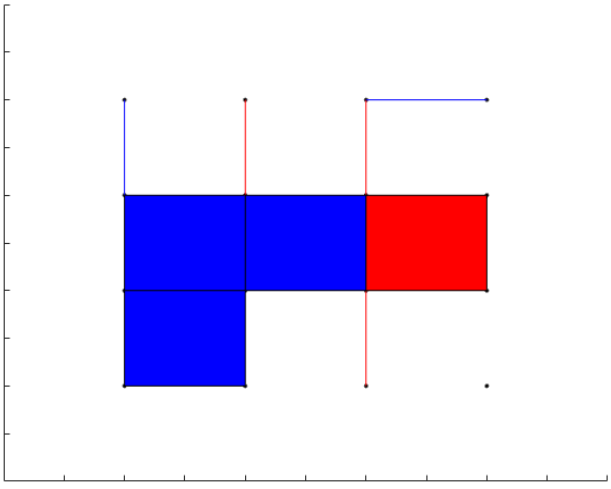

Figure 2: 3x3 game with blue player as the trainer (player 2) and red player (player 1)

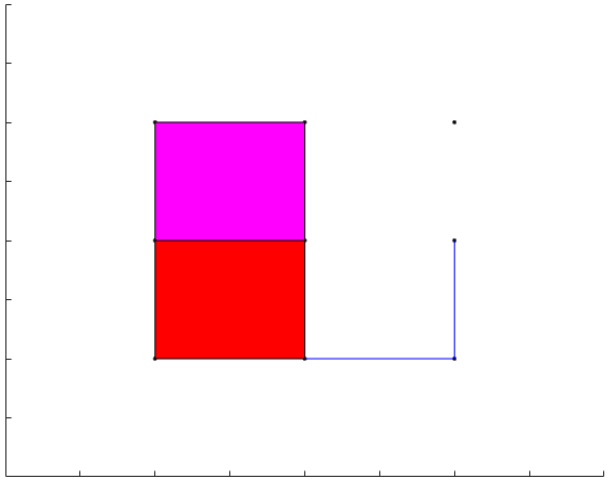

Figure 3: 2x2 game with pink player as the evaluator (player 2) and red player (player 1)

The game ends when no more lines can be created. The points are then summed for each player and the player with the most points wins the game.

Approach

The approach for solving this game revolves around the ability to determine all states of the game given any combination of moves made and not made in the play space. With the 2x2 game, the states of the game space can be identified as a binary string corresponding position where the line in drawn regardless of which player drew it. This results in 2^12, or 4096 possible states from 12 possible moves.

However, for the 3x3 game the number of possible states jumps to 2^24, or 16,777,216 possible states for 24 moves. To combat this issue of expanding states, the game was treated as four interconnected games.

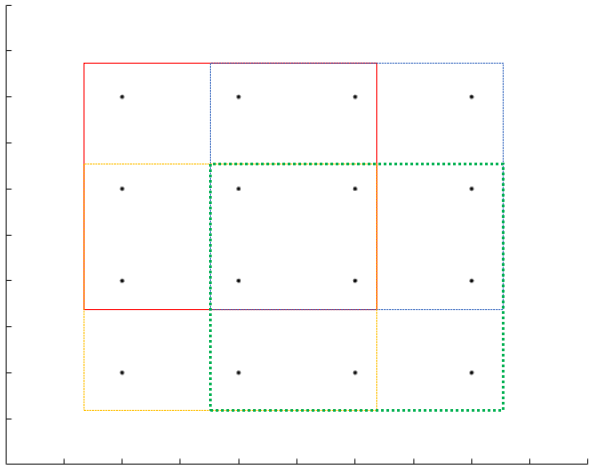

Figure 4: The overlap of the 4 2x2 games on the 3x3 board

This reduced the number of states down from 2^24 to approximately 4*2^12 with shared lines. This let the machine play 4 separate games each with their own set of Q values that could then be compared to determine the overall state of the game and the best possible move to make.

Q Learner

Q Learning is a form of Machine Learning that iterates over a Monte Carlo Control method. This method involves the usage of a policy improvement and policy evaluation to determine the outputs of decision and improving based on an epsilon soft policy. The soft policy that was used in this game was an epsilon-greedy, a policy that selects the next move based on maximizing the possible outcome from a "Q" table. This Q table is a set of values that is determined by a reward function and the state-action pair that resulted in it.

For the 2x2 game, the Q table was initially a 4096 by 12 table. This was able to be reduced to a 4096x1 array by assigning the binary state encoding to a lookup key. The same was done for the 3x3 game resulting in a 4096x4 Q table. The game itself had player one as the Q learner and for every 100 games played, player two would receive the Q table from player one. To make a move the state action pair calls the Q table and relays a state action pair that results in Q values. The maximum of these Q values will result in that move being picked. The Learner is then trained for a selected number of games (in this case 100, 300, 500, 600, 1000, 5000, and 10,000 games) and then played against a bot that will make randomized moves.

Functional Approximation

Bootstrapping the training data and Q table from the Q-Learner, the FA, using Tensor Flow, will then train on the data set and approximate a table to use as a predictor for future state-action combos. This particular Functional Approximator was made of three layers initializing with the same number of neurons as state action pairs and half number of neurons per successive layer. This design was inspired by Matthew Deakos. (Deakos, 2017) and, in-part, helped designed by Corbyn Yhap and Eli Grubb of the

University of Maryland, College Park. Upon entering the game sequence, the program will call the predictor with a look ahead call of current state and all possible actions and receive back Q values associated with the actions. Much of the post processing of the data is very similar to that of the Q Learner. The benefit over the Q Learner by using the FA is that now the results are with respect to the actions rather than the states. This is a benefit because the larger the game, the larger the states and with respect to the dots and boxes the game increases from the 2x2 with 4096 states to the 3x3 game with a minimum around 16,384 using the four 2x2 interconnected games. This would mean that less training (less known state action pairs) will be needed to achieve better results.

Results

The results of the games are divided into four categories: Q Learner 2x2, Q Learner 3x3, Functional Approximator 2x2, and Functional Approximator 3x3. Each one was trained at 100, 300, 500, 600, 1,000, 5,000 and 10,000 games and then played for 5 sets of 100 games. The averages, highs and lows are then plotted. Finally, a benchmark bar chart displays the performances (of the Q learner and FA) in both the 2x2 and 3x3 at different training levels.

Q-Learner

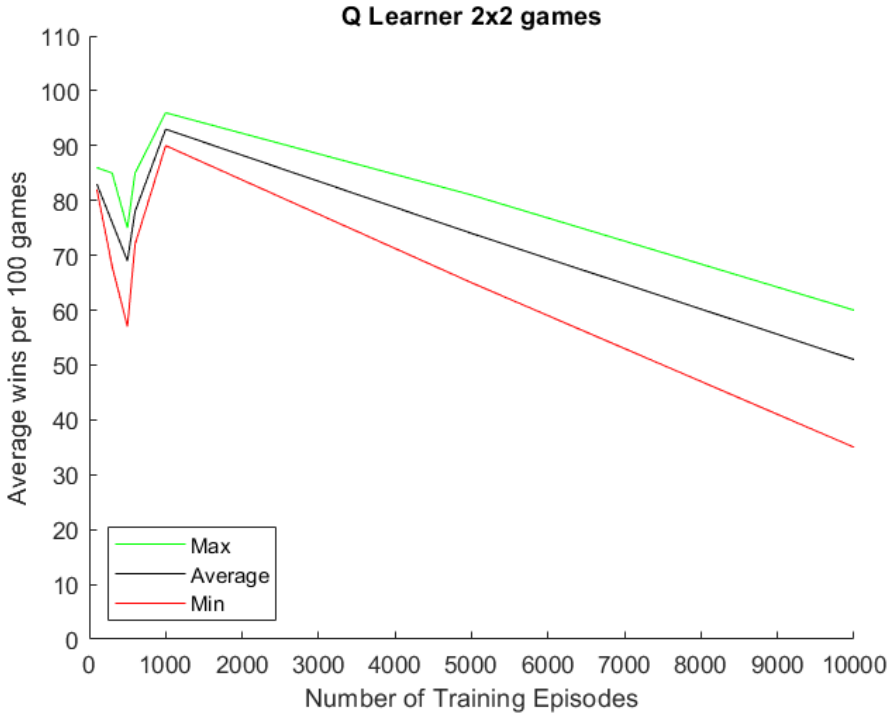

Figure 5: Q Learner 2x2 game against evaluator for 500 games at each training episode

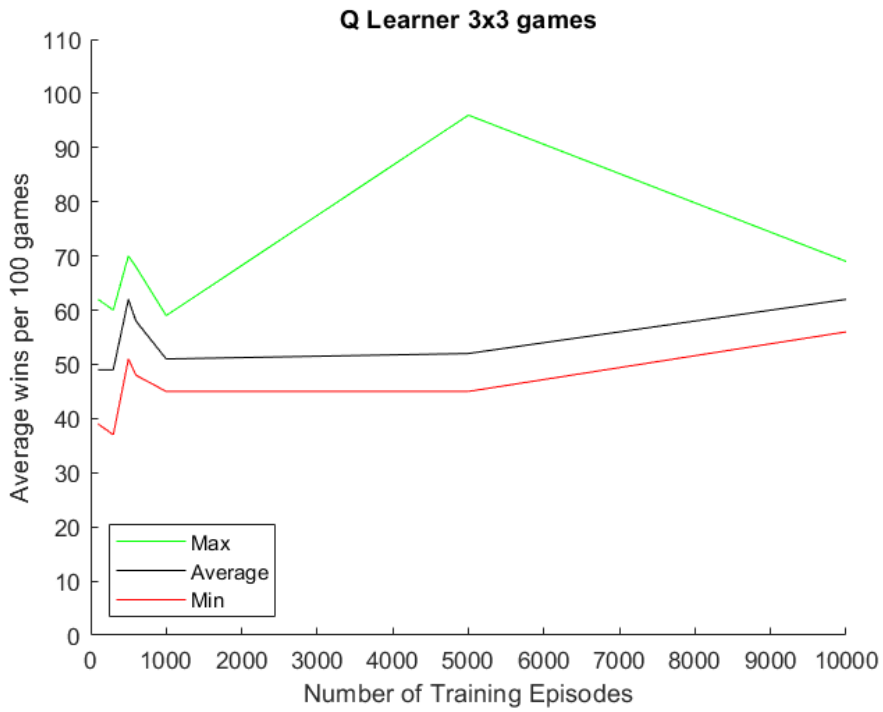

Figure 6: Q Learner 3x3 game against evaluator for 500 games at each training episode

Both the 2x2 and 3x3 demonstrate nominal performance around the 1000 games training mark. As the number of training points increases the margins of successful play (difference between the maximum wins per 100 games and minimum number of wins per 100 games) also tend to increase. Comparatively, between the 2x2 and the 3x3 via the number of successful games, the 2x2 outperforms the 3x3 with the 2x2 winning 20% more games than the 3x3 until approximately the 10,000 training episodes mark. Additionally, it is observed that the 3x3 begins and upward trend of winning games as it gets closer to the number of possible states in the Q table for the 3x3.

Functional Approximator

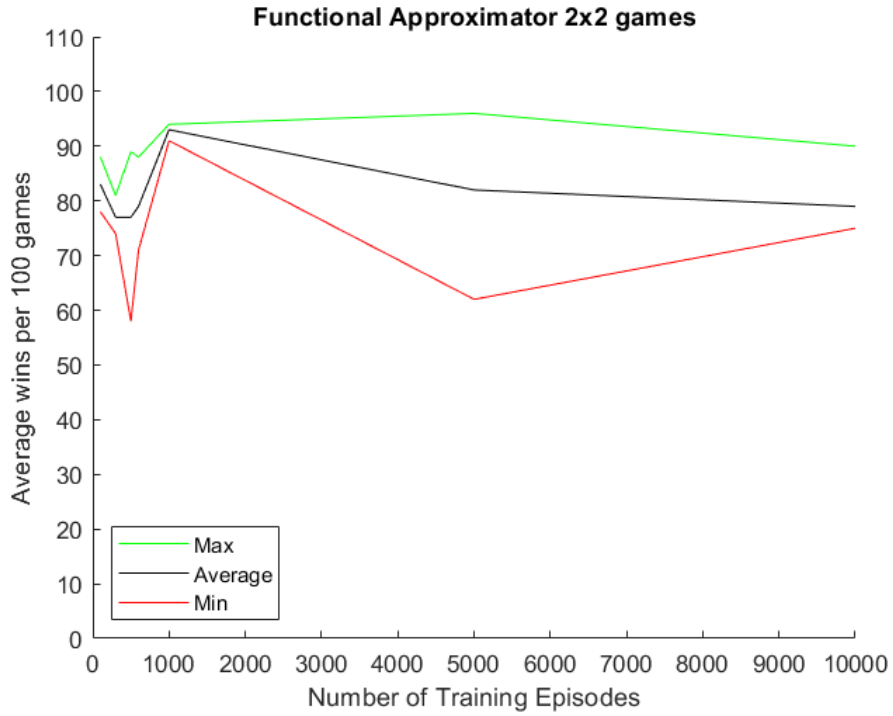

Figure 7: Functional Approximator 2x2 game against evaluator for 500 games at each training episode

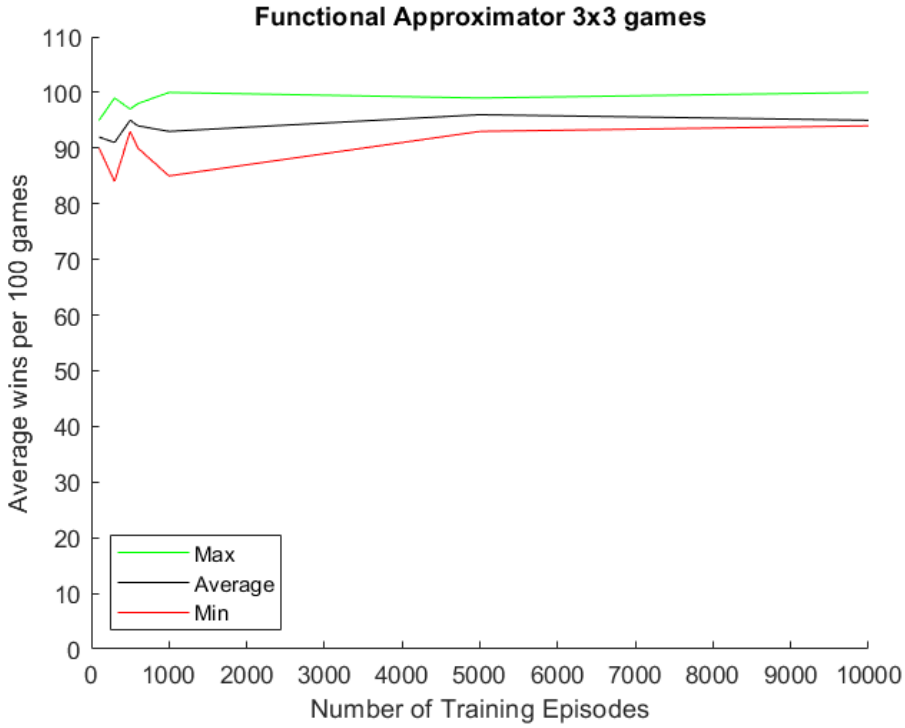

Figure 8: Functional Approximator 3x3 game against evaluator for 500 games at each training episode

As with the Q Learner, the FA appears to perform best about the 1000 training episode mark. The margin of successful play, except for the 5000 training episodes, is relatively small suggesting that the FA wins games more consistently. Comparing the results of the 2x2 and the 3x3, the 3x3 game outperforms the 2x2 game in terms of average wins.

It is believed that the 3x3 outperforms the 2x2 in the functional approximator due to the sheer number of states that occur in the 3x3 games lending to the strength of the FA (the reliance of action over state). This allows the FA to make more calculated moves in the random game resulting in less randomly made/player two boxes. This could qualify as a saturation in performance for the 3x3 where as the saturation for the 2x2 likely occurs closer to the 5000-training episode mark due to the number of states. This is support by the chart as the FA in the 2x2 was able to achieve a 100% game win performance, despite also having a low margin of winning, likely due to over training occurring after 4096 games.

Benchmarking

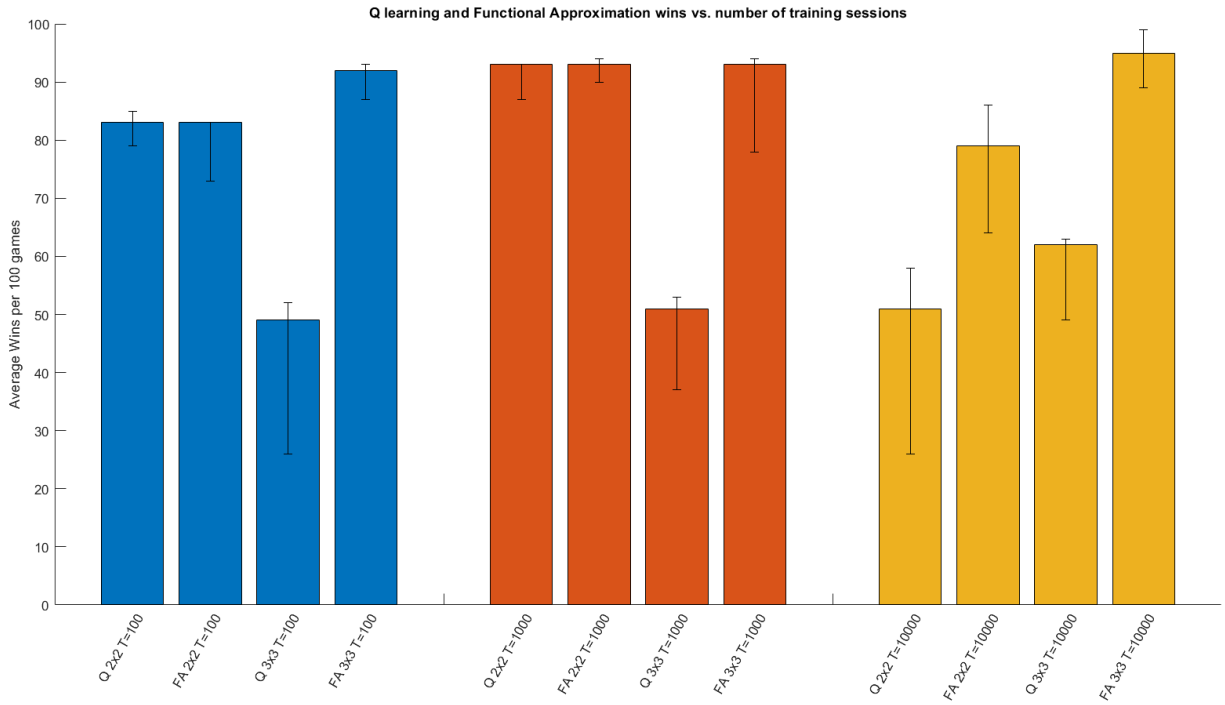

Figure 9: Q Learner and Functional Approximator for 2x2 and 3x3 games against evaluator for 500 games at training episodes 100, 1000, 10000 with margins of successful play

The benchmark chart, like the results from the Q Learner and the FA, demonstrate an increase in learning across the board until the 10,00 mark, with the exception of the FA for the 3x3. Comparatively, the FA outperforms the Q Learner in the 3x3 in every case; however, the Q Learner and the FA for the 2x2 perform roughly the same, through the FA performs slightly better overall, until the 10,000-training episode mark when the Q Learner sees a drop off in performance.

Analysis and Future Work

Drop in Performance

It is believed that the 2x2 performs well until the 5000-training episode due to overtraining starting to occur. Since there are only 4096 states, training the Q Learner for 5000+ training episodes, the Q Learner will begin to overwrite some actions that were beneficial. This does not occur as harshly for the FA since the FA is more dependent on action rather than state. With regard to the 3x3, it makes sense that the 3x3 in either the Q Learner or the FA improves as the number of training sessions increases as there are greater than 10,000 possible states.

100 Game Discrepancy

It can be seen across all the bots used to play, that at the 100-training episode mark performs better than when more training episodes are used, up to a certain number of episodes. This is likely due to how the Q Learning training occurs and is evaluated. Since the Q Learner plays against itself, the first 100 games it plays against another Q Learner with a Q table of all zeros, player 2. The player 2 Q Learner, when calling the Q table, chooses moves in a similar manor that the random move bot makes since the Q table, for player 2, is not updated till the 101st game. This results in the player 1 Q Learner learning that the moves that are made are random, instead of intentionally trying to maximize the number of points.

Improving Overall Performance

Additionally, it was noted that the argument for choosing the Q value for the next action was that the maximum Q value was selected, and the corresponding action was chosen with it. In the case that there were multiple Q's with the same value the first Q value is always selected in both the Q Learner and the FA. This leads to issues when training the data, as more training episodes are used and results in the game being played right to left almost every time for player 1.

There are three ways that this could be solved: First is that the actions pertaining to the Q values can be shuffled prior to them being chosen. Second, choice of the Q value can be redistributed such that the actions that have equal Q values will redistribute the percent chance that the action is chosen. Example: where there are 5 Q numbers and Q numbers 1 and 3 have the same value and are greater than the rest of the 3 values in the set. In the current version of choosing the action, the 3rd Q number is evaluated like the other non-max Q values of the set; such that, Q number 1 has, say, a 95% chance of being chosen and the and the rest have a 1% chance. The redistribution would make it such that Q numbers 1 and 3 would both have 47.5% chance of being chose and the rest with 1% chance. Finally, the Q Leaner can be trained using a greedy bot rather than against a Q Learner bot. This forces the Q Learner to play more aggressively and try new moves across the board since the greedy algorithm will not allow the Q Learner to make moves to create boxes without drawing the attention of the greedy bot.

Future work

For the 2x2, limiting the number of training sessions for either the Q Learner or the FA would likely result in greater performance. The Q learner in in the 3x3 could greatly be improved by considering the issues noted in the "Drop in Performance" and "Improving Overall Performance" sections. The 3x3 could best be improved by how the Q value/Action policy is determined and increasing the number of training episodes closer to that of the number of states. Additionally, using the greedy bot to train the 3x3 results in greater variance in moves that complete boxes rather than setup boxes to be completed if the

random bot does not complete them first. Finally, if the Epsilon value for the Epsilon-greedy policy is increased the Q Learner may learn to fill boxes created by the random moves more often than attempting to create setup boxes and then try to fill them in.

Another part of future work revolves around how this program was developed. The main program is in MATLAB and the FA is written in Python. Initially, this was not the indentation as the newest version of MATLAB (as of May of 2019), included a newly released tool box for Reinforcement Q learning. Unfortunately, due to how new the toolbox is, there is not much documentation to develop environments and train a Q agent. Revisiting this program when there is more documentation on the toolbox would allow for the development of a much faster program as the there will be no need for reading and writing to files to be used in another program.

References

Deakos, M. (2017). *MLND Capstone Project A Deep Reinforcement Learning Approach to "Dots and Boxes".*

Seita, D. (2016, October 31). *Going Deeper Into Reinforcement Learning: Understanding Q-Learning and Linear Function Approximation*. Retrieved from https://danieltakeshi.github.io/2016/10/31/going-deeper-into-reinforcement-learning-understanding-q-learning-and-linear-function-approximation/

Sofge, D. (2019). Q-Learning and Actor-Critic Models. College Park, Maryland.

Sutton, R. S., & Barto, G. A. (1998). *Reinforcement Learning:.* Cambridge: MIT Press.

Wikipedia. (2019, April 28). *Q-learning*. Retrieved from https://en.wikipedia.org/wiki/Q-learning

Code

Matlab

Main

```
close all

%Create Points
clc
clearvars -except layers_1

%Initalize all the Functional Approximation Data sets
DLIn=[];
DLState=[];
DLStateSet=[];
DLAction=[];
DLReward=[];
DLQVals=[];

DLStateEpisode=[];
DLStateSetEpisode=[];
DLActionEpisode=[];
DLRewardEpisode=[];
DLEpisodeSize=[];
DLQValsEpisode=[];

%Make all the State possibilities for the 2x2 game
States=0:1:2^12-1;

States=States.';

StatesBin=dec2bin(States);
SBSA=[];

for ii=1:length(States)
   StatesBinSep=str2double(regexp(convertCharsToStrings(StatesBin(ii,:)),'\d','match'));
   SBSA=[SBSA;StatesBinSep];
end

%%
%Make the Game based on the number of boxes the user enters
Size=input('Input size: ')
Points=[];
for ii=1:1:Size(1)+1

   for jj=1:1:Size(2)+1
      Points=[Points,[ii;jj]];
   end
end

TN=input('Number of Training Sessions: ')
for Training=1:1:TN
   hold on
      figure(1)

      %Create the dots
      for kk=1:1:length(Points)
         plot(Points(1,kk),Points(2,kk), 'k.')
      end
```

```
%    %Formatting grid
ax=gca;
axis([0,Size(1)+2,0,Size(1)+2]);
grid off;
ax.XTickLabel=[];
ax.YTickLabel=[];

%%
%Total Possible Moves
TPM=[];
MvN=1;
CTPList=[];
TPLsit=[];
invTPMCheck=[0 0;0 0];
for qq=1:1:length(Points)
   CTP=Points(:,qq);


   for rr=1:1:length(Points)
      TP=Points(:,rr);

      if (abs(CTP(1)-TP(1))==1 && abs(CTP(2)-TP(2))==0)|| (abs(CTP(1)-TP(1))==0 && abs(CTP(2)-TP(2))==1)
         dubcheck=[CTP,TP];

         if any(all(all(bsxfun(@eq,invTPMCheck,dubcheck))))==0
            TPM(:,:,MvN)=[CTP,TP];
            invTPMCheck(:,:,MvN)=[TP,CTP];
            MvN=MvN+1;
         end
      end
   end
end
%%

%Initalize Values for the game
MaxMoves=length(TPM);
AllowedMoves=TPM;
CM=1;
ww=1;
Player=1;
Ply=1;
Player1Points=0;
P1PP=0;
P2PP=0;
Player2Points=0;
UsedNN=[];
UsedNodes=[];
BoxWalls=zeros(1,Size(1)*Size(2));
Player1Total=0;
Player2Total=0;
FA=0;

%Create the Vertical and Horizontal Value lines
VertVals=[];
HorzVals=[];
VertLow=1;
VertHigh=MaxMoves;
HorzLow=2;
```

```
HorzDif=Size(1)*2-1;
HorzMid=HorzLow+2;
HorzMidG=[];
HorzHigh=HorzLow+HorzDif;
VertMid=VertLow+HorzDif+2;
VertMidG=[];
for ii=1:1:Size(1)

   for jj=1:1:Size(1)-1

      HorzMidG=[HorzMid;HorzMidG];
      HorzMid=HorzMid+2;
   end

   for ll=1:1:Size(1)-1

      VertMidG=[VertMidG,VertMid];
      VertMid=VertMid+HorzDif+2;
   end

   VertVals=[[VertLow,VertMidG,VertHigh];VertVals];
   HorzVals=[HorzVals,[HorzHigh;HorzMidG;HorzLow]];
   VertLow=VertLow+2;
   VertHigh=VertHigh-1;
   HorzLow=HorzLow+HorzDif+2;
   HorzHigh=HorzLow+HorzDif;
   HorzMid=HorzLow+2;
   VertMid=VertLow+HorzDif+2;
   HorzMidG=[];
   VertMidG=[];


end

%Organize the Horizonal and Veritcal line values
HorzVals=flipud(HorzVals);
LVV=size(VertVals);
VertVals(:,LVV(2))=flipud(VertVals(:,LVV(2)));
VertVals=flipud(VertVals);



%Creating Boxes
Boxes=[];
for ii=1:1:length(HorzVals)-1
   for jj=1:1:length(VertVals)-1
      Box=[HorzVals(ii,jj),HorzVals(ii+1,jj),VertVals(ii,jj),VertVals(ii,jj+1)];
      Boxes=[Boxes;Box];
   end

end
BoxesLeft=Boxes;

%Initalize the 2x2 games inside the 3x3 games
if Size==[3,3]
   QuarterSet1=unique([Boxes(1,:),Boxes(2,:),Boxes(4,:),Boxes(5,:)]);
   QuarterSet2=unique([Boxes(4,:),Boxes(5,:),Boxes(7,:),Boxes(8,:)]);
   QuarterSet3=unique([Boxes(2,:),Boxes(3,:),Boxes(5,:),Boxes(6,:)]);
   QuarterSet4=unique([Boxes(5,:),Boxes(6,:),Boxes(8,:),Boxes(9,:)]);
```

```
    QuarterSets=[QuarterSet1;QuarterSet2;QuarterSet3;QuarterSet4];

  else
    QuarterSets=unique([Boxes(1,:),Boxes(2,:),Boxes(4,:),Boxes(3,:)]);
  end
  AllMoves=1:MaxMoves;
  [State,StateSet]=getState(UsedNN,SBSA,QuarterSets);

  %Initialize Q table for both Player 1 and 2 or initialize Q table

  if Training==1
    Q1=zeros(length(SBSA),length(StateSet));
    Qs=Q1;
    Q2=Q1;
  else
    Q1=QOut;
    Qs=QOut;

    if mod(Training,100)==1
      Q2=Q1;
      Training
    else
      Q2=QOut;
    end

  end

  %    input('')
  %Main Program
  while CM~=MaxMoves+1
    EG=0;

    %Update states and the remaining Moves
    RMoves=setdiff(AllMoves,UsedNN);
    [State,StateSet]=getState(UsedNN,SBSA,QuarterSets);


    if Player==1
      %Player1
%       Update QTable and Select Next Move

[Q1,P1PP,RewardOut1]=QUpdate(Q1,Player1Points,Player2Points,State,StateSet,EG,P1PP,AllowedMoves,UsedNN,SBSA,Quarte
rSets,Boxes,Size,BoxWalls,TPM,UsedNodes,Player1Points);

[Move,TPMNo]=RLearner(AllowedMoves,TPM,UsedNodes,UsedNN,Boxes,Size,Player1Points,Player2Points,BoxWalls,State,Stat
eSet,SBSA,QuarterSets,Player1Points,Q1,0,FA);

      %Update the Per Turn Functional Approximator values
      if Size==[3,3]
        StateR=SBSA(StateSet,:);
        StateR=StateR(:).';
        DLState=[DLState;StateR];
      else
        DLState=[DLState;State];
      end
      DLStateSet=[DLStateSet;StateSet];
      DLAction=[DLAction;TPMNo];
      DLReward=[DLReward;RewardOut1];
```

```
        if Size==[3,3]
            DLQVals=[DLQVals;sum(Q1(StateSet))];
        end

        %Update Visuals
        plot(AllowedMoves(1,:,Move),AllowedMoves(2,:,Move),'r')
        Color=[1,0,0];
    else
        %Player2

        %Random Bot
%           [Move,TPMNo]=random(AllowedMoves,TPM);


        %Greedy Bot
%
[Move,TPMNo]=Greedy(AllowedMoves,TPM,UsedNodes,UsedNN,Boxes,Size,Player2Points,Player1Points,BoxWalls,State,StateS
et,SBSA,QuarterSets,Player2Points,Q2,0,FA);


        %Q Bot

[~,P2PP,~]=QUpdate(Q2,Player2Points,Player1Points,State,StateSet,EG,P2PP,AllowedMoves,UsedNN,SBSA,QuarterSets,Boxes,Si
ze,BoxWalls,TPM,UsedNodes,Player2Points);

[Move,TPMNo]=RLearner(AllowedMoves,TPM,UsedNodes,UsedNN,Boxes,Size,Player2Points,Player1Points,BoxWalls,State,Stat
eSet,SBSA,QuarterSets,Player2Points,Q2,0,FA);

        %Update Visuals
        plot(AllowedMoves(1,:,Move),AllowedMoves(2,:,Move),'b')
        Color=[0,0,1];
    end


    %Update the Availible Moves
    UsedNN=[UsedNN,TPMNo];
    UsedNodes(:,:,CM)= AllowedMoves(:,:,Move);
    AllowedMoves(:,:,Move)=[];

    %Check to see if it was a box and update the Drawing, Scores and
    %Player turn
    [YIB,WhichB]=isBox(UsedNodes,UsedNN,TPMNo,BoxesLeft,Size,BoxWalls);
    if sum(YIB)~=0
        for ii=1:1:length(YIB)

            if YIB(ii)==1
                fillx=[];
                filly=[];
                %
                for kk=1:1:5
                    fillx=[fillx,TPM(1,:,WhichB(ii,kk))];
                    filly=[filly,TPM(2,:,WhichB(ii,kk))];


                end
                minx=min(fillx);
                maxx=max(fillx);
                miny=min(filly);
```

```
            maxy=max(filly);
            fzx=[minx,minx,maxx,maxx,minx];
            fzy=[miny,maxy,maxy,miny,miny];
            %                 input('FillSquare ')
            fill(fzx,fzy,Color)



            if  Player==1
               Player1Points=Player1Points+1;
            else
               Player2Points=Player2Points+1;
            end
          end
        end
      else
        Ply=Ply+1;
        Player=mod(Ply,2);
      end


      %          input(' ')
          pause(.001)
      CM=CM+1;
    end
    hold off

    % disp('Game Over')
    % disp(' ')

    if Player1Points>Player2Points
      %    disp('Player 1 Wins')
      %    Player1Points
      %    Player2Points

    elseif Player2Points>Player1Points
      %    disp('Player 2 Wins')
      %    Player1Points
      %    Player2Points
    else
      %    disp('Tie Game')
    end
    %End the Game and Update all State and Q Values
    EG=1;
    [State,StateSet]=getState(UsedNN,SBSA,QuarterSets);

[Q1,P1PP,RewardOut1]=QUpdate(Q1,Player1Points,Player2Points,State,StateSet,EG,P1PP,AllowedMoves,UsedNN,SBSA,Quarte
rSets,Boxes,Size,BoxWalls,TPM,UsedNodes,Player1Points);

[~,P2PP,RewardOut2]=QUpdate(Q2,Player2Points,Player1Points,State,StateSet,EG,P2PP,AllowedMoves,UsedNN,SBSA,QuarterS
ets,Boxes,Size,BoxWalls,TPM,UsedNodes,Player2Points);


    %Function Approximation Value Development
    if Size==[3,3]
      StateR=SBSA(StateSet,:);
      StateR=StateR(:).';
      DLState=[DLState;StateR];
```

```
    else
        DLState=[DLState;State];
    end
    DLStateSet=[DLStateSet;StateSet];
    DLAction=[DLAction;TPMNo];
    DLReward=[DLReward;RewardOut1];

    if Size==[3,3]
        DLQVals=[DLQVals;sum(Q1(StateSet))];
    else
        DLQVals=[DLQVals;Q1(DLStateSet)];
    end

    DLStateEpisode=[DLStateEpisode;DLState];
    DLEpisodeSize=[DLEpisodeSize;length(DLStateSet)];
    DLStateSetEpisode=[DLStateSetEpisode;DLStateSet];
    DLActionEpisode=[DLActionEpisode;DLAction];
    DLRewardEpisode=[DLRewardEpisode;DLReward];
    DLQValsEpisode=[DLQValsEpisode;DLQVals];

    DLState=[];
    DLStateSet=[];
    DLAction=[];
    DLReward=[];
    DLQVals=[];

    %Set the Q Value for the next round
    QOut=Q1;
    clf(figure(1))
end
%%
%Function Approximation Value Concatonation
QFA=[];
DLSA=[DLStateEpisode,DLActionEpisode];


%%
%Choose whether to run the Functional Approximator
FA=input('Enter 1 to use Function approximation else, press enter: ')
gamerun=0;
Player1WinSet=[];

while gamerun<5

    if FA==1

        %Write out State action pairs to Python to create the Approximator
        Lernt=writeoutDL(DLSA,DLQValsEpisode,Size);
        disp('Starting')

        %Initialize for the Evaluation gameplay
        QGame=Q1;
        Player1PointTotal=0;
        Player2PointTotal=0;
        Player1WinTotal=0;
        Player2WinTotal=0;
        TieGames=0;

        %Play the Evaluation Game set
```

```
[Player1PointsTotal,~,Player1WinTotal,Player2WinTotal,TieGames]=GameOn(QGame,Size,Player1PointTotal,Player2PointTotal,
Player1WinTotal,Player2WinTotal,TieGames,FA)
      close all

   else

      %Initialize for the Evaluation gameplay
      FA=0;
      QGame=Q1;
      Player1PointTotal=0;
      Player2PointTotal=0;
      Player1WinTotal=0;
      Player2WinTotal=0;
      TieGames=0;

      %Play the Evaluation Game set

[~,~,Player1WinTotal,Player2WinTotal,TieGames]=GameOn(QGame,Size,Player1PointTotal,Player2PointTotal,Player1WinTotal,
Player2WinTotal,TieGames,FA)
      close all

   end

   Player1WinSet=[Player1WinSet,Player1WinTotal];
   gamerun=gamerun+1;

end
close all

%Output the Evaluation Set
Player1WinSet=Player1Winset


%%
%%%%%%
%Game%
%%%%%%


function
[Player1PointTotal,Player2PointTotal,Player1WinTotal,Player2WinTotal,TieGames]=GameOn(Q,Size,Player1PointTotal,Player2P
ointTotal,Player1WinTotal,Player2WinTotal,TieGames,FA)

States=0:1:2^12-1;

States=States.';

StatesBin=dec2bin(States);
SBSA=[];

for ii=1:length(States)
   StatesBinSep=str2double(regexp(convertCharsToStrings(StatesBin(ii,:)),'\d','match'));
   SBSA=[SBSA;StatesBinSep];
end

Points=[];
for ii=1:1:Size(1)+1
```

```
    for jj=1:1:Size(2)+1
        Points=[Points,[ii;jj]];
    end
end
for Game=1:1:100
    hold on
    figure(1)
    for kk=1:1:length(Points)
        plot(Points(1,kk),Points(2,kk), 'k.')
    end

    %Formatting grid
    ax=gca;
    axis([0,Size(1)+2,0,Size(1)+2]);
    grid off;
    ax.XTickLabel=[];
    ax.YTickLabel=[];

    %%
    %Total Possible Moves
    TPM=[];
    MvN=1;
    CTPList=[];
    TPLsit=[];
    invTPMCheck=[0 0;0 0];
    for qq=1:1:length(Points)
        CTP=Points(:,qq);


        for rr=1:1:length(Points)
            TP=Points(:,rr);

            if (abs(CTP(1)-TP(1))==1 && abs(CTP(2)-TP(2))==0)|| (abs(CTP(1)-TP(1))==0 && abs(CTP(2)-TP(2))==1)
                dubcheck=[CTP,TP];

                if any(all(all(bsxfun(@eq,invTPMCheck,dubcheck))))==0
                    TPM(:,:,MvN)=[CTP,TP];
                    invTPMCheck(:,:,MvN)=[TP,CTP];
                    MvN=MvN+1;
                end
            end
        end
    end
    %%

    MaxMoves=length(TPM);
    AllowedMoves=TPM;
    CM=1;
    ww=1;
    Player=1;
    Ply=1;
    Player1Points=0;
    P1PP=0;
    P2PP=0;
    Player2Points=0;
    UsedNN=[];
    UsedNodes=[];
    BoxWalls=zeros(1,Size(1)*Size(2));
```

```
%OusideVals
VertVals=[];
HorzVals=[];
VertLow=1;
VertHigh=MaxMoves;
HorzLow=2;
HorzDif=Size(1)*2-1;
HorzMid=HorzLow+2;
HorzMidG=[];
HorzHigh=HorzLow+HorzDif;
VertMid=VertLow+HorzDif+2;
VertMidG=[];
for ii=1:1:Size(1)

  for jj=1:1:Size(1)-1

    HorzMidG=[HorzMid;HorzMidG];
    HorzMid=HorzMid+2;
  end

  for ll=1:1:Size(1)-1

    VertMidG=[VertMidG,VertMid];
    VertMid=VertMid+HorzDif+2;
  end

  VertVals=[[VertLow,VertMidG,VertHigh];VertVals];
  HorzVals=[HorzVals,[HorzHigh;HorzMidG;HorzLow]];
  VertLow=VertLow+2;
  VertHigh=VertHigh-1;
  HorzLow=HorzLow+HorzDif+2;
  HorzHigh=HorzLow+HorzDif;
  HorzMid=HorzLow+2;
  VertMid=VertLow+HorzDif+2;
  HorzMidG=[];
  VertMidG=[];


end


HorzVals=flipud(HorzVals);
LVV=size(VertVals);
VertVals(:,LVV(2))=flipud(VertVals(:,LVV(2)));
VertVals=flipud(VertVals);



%Creating Boxes
Boxes=[];
for ii=1:1:length(HorzVals)-1
  for jj=1:1:length(VertVals)-1
    Box=[HorzVals(ii,jj),HorzVals(ii+1,jj),VertVals(ii,jj),VertVals(ii,jj+1)];
    Boxes=[Boxes;Box];
  end

end
BoxesLeft=Boxes;
if Size==[3,3]
```

```
    QuarterSet1=unique([Boxes(1,:),Boxes(2,:),Boxes(4,:),Boxes(5,:)]);
    QuarterSet2=unique([Boxes(4,:),Boxes(5,:),Boxes(7,:),Boxes(8,:)]);
    QuarterSet3=unique([Boxes(2,:),Boxes(3,:),Boxes(5,:),Boxes(6,:)]);
    QuarterSet4=unique([Boxes(5,:),Boxes(6,:),Boxes(8,:),Boxes(9,:)]);

    QuarterSets=[QuarterSet1;QuarterSet2;QuarterSet3;QuarterSet4];

  else
    QuarterSets=unique([Boxes(1,:),Boxes(2,:),Boxes(4,:),Boxes(3,:)]);
  end
  AllMoves=1:MaxMoves;
  [State,StateSet]=getState(UsedNN,SBSA,QuarterSets);


  %Main Program
  while CM~=MaxMoves+1
    EG=0;

    RMoves=setdiff(AllMoves,UsedNN);
    [State,StateSet]=getState(UsedNN,SBSA,QuarterSets);


    if Player==1
      %Player1
      %       [Move,TPMNo]=random(AllowedMoves,TPM);

[Move,TPMNo]=RLearner(AllowedMoves,TPM,UsedNodes,UsedNN,Boxes,Size,Player1Points,Player2Points,BoxWalls,State,Stat
eSet,SBSA,QuarterSets,Player1Points,Q,1,FA);
      plot(AllowedMoves(1,:,Move),AllowedMoves(2,:,Move),'r')
      Color=[1,0,0];
    else
      %Player2

      %GreedyBot
      %
[Move,TPMNo]=Greedy(AllowedMoves,TPM,UsedNodes,UsedNN,Boxes,Size,Player2Points,Player1Points,BoxWalls,State,StateS
et,SBSA,QuarterSets,Player2Points,Q,0,FA);

      %RandomBot
      [Move,TPMNo]=random(AllowedMoves,TPM);
      plot(AllowedMoves(1,:,Move),AllowedMoves(2,:,Move),'b')
      Color=[1,0,1];
    end

    UsedNN=[UsedNN,TPMNo];
    UsedNodes(:,:,CM)= AllowedMoves(:,:,Move);
    AllowedMoves(:,:,Move)=[];


    [YIB,WhichB]=isBox(UsedNodes,UsedNN,TPMNo,BoxesLeft,Size,BoxWalls);
    if sum(YIB)~=0
      for ii=1:1:length(YIB)

        if YIB(ii)==1
          fillx=[];
          filly=[];
          %
          for kk=1:1:5
            fillx=[fillx,TPM(1,:,WhichB(ii,kk))];
```

```
            filly=[filly,TPM(2,:,WhichB(ii,kk))];


        end
        minx=min(fillx);
        maxx=max(fillx);
        miny=min(filly);
        maxy=max(filly);
        fzx=[minx,minx,maxx,maxx,minx];
        fzy=[miny,maxy,maxy,miny,miny];
        %               input('FillSquare ')
        fill(fzx,fzy,Color)



        if  Player==1
          Player1Points=Player1Points+1;
        else
          Player2Points=Player2Points+1;
        end
      end
    end
  else
    Ply=Ply+1;
    Player=mod(Ply,2);
  end


  pause(.001)
  CM=CM+1;
end
hold off


if Player1Points>Player2Points

  Player1Points;
  Player2Points;
  Player1WinTotal=Player1WinTotal+1;
  Player2WinTotal=Player2WinTotal;
elseif Player2Points>Player1Points

  Player1Points;
  Player2Points;
  Player2WinTotal=Player2WinTotal+1;
  Player1WinTotal=Player1WinTotal;
else
  TieGames=TieGames+1;
end

Player1PointTotal=Player1PointTotal+Player1Points;
Player2PointTotal=Player2PointTotal+Player2Points;

clf(figure(1))

end

end
```

RLearner

```
function
[Move,TPMNo,State]=RLearner(AllowedMoves,TPM,UsedNodes,UsedNN,Boxes,Size,MyPoints,EnemyPoints,BoxWalls,State,Stat
eSet,SBSA,QuarterSets,StatePoints,Q,Game,FA)
[R,C,M]=size(AllowedMoves);
ChoiceSet=[];


%Creating a look ahead set of State Action Pairs
%     disp('Look Ahead')
for ii=1:1:M
   TempUsedNN=UsedNN;
   CurMov=AllowedMoves(:,:,ii);
   TPMNoC = find(arrayfun(@(x) isequal(TPM(:,:,x),CurMov),1:size(TPM,3)));
   TempUsedNN=[TempUsedNN,TPMNoC];
   [YIB,~]=isBox(UsedNodes,TempUsedNN,TPMNoC,Boxes,Size,BoxWalls);
   BoM=sum(YIB);
   [TempState,TempStateset]=getState(TempUsedNN,SBSA,QuarterSets);
   TempStateset=TempStateset.';
   ChoiceSet=[ChoiceSet;[TPMNoC,BoM,StatePoints,TempStateset]];
end

%Policy
if FA==1
   Choice=QPolicy2(ChoiceSet,SBSA,Q,StatePoints,Game,State);
else
   Choice=QPolicy(ChoiceSet,SBSA,Q,StatePoints,Game,State);
end
%Where is the choice value in the lineup?
ChosenVal=find(Choice==ChoiceSet);

%Greedy
%     ChosenVal=find(ChoiceSet(:,2)==max(ChoiceSet(:,2)))

Move=ChosenVal(1);
CurMov=AllowedMoves(:,:,Move);
TPMNo = find(arrayfun(@(x) isequal(TPM(:,:,x),CurMov),1:size(TPM,3)));

TPMNoSupdate=TPMNo;

end
```

Greedy

```
function
[Move,TPMNo,State]=Greedy(AllowedMoves,TPM,UsedNodes,UsedNN,Boxes,Size,MyPoints,EnemyPoints,BoxWalls,State,State
Set,SBSA,QuarterSets,StatePoints,Q,Game,FA)
[R,C,M]=size(AllowedMoves);
ChoiceSet=[];
%     disp('Look Ahead')
for ii=1:1:M
   TempUsedNN=UsedNN;
   CurMov=AllowedMoves(:,:,ii);
   TPMNoC = find(arrayfun(@(x) isequal(TPM(:,:,x),CurMov),1:size(TPM,3)));
   TempUsedNN=[TempUsedNN,TPMNoC];
   [YIB,~]=isBox(UsedNodes,TempUsedNN,TPMNoC,Boxes,Size,BoxWalls);
   BoM=sum(YIB);
   [TempState,TempStateset]=getState(TempUsedNN,SBSA,QuarterSets);
```

```
    TempStateset=TempStateset.';
    ChoiceSet=[ChoiceSet;[TPMNoC,BoM,StatePoints,TempStateset]];
end

% %Policy
% if FA==1
%     Choice=QPolicy2(ChoiceSet,SBSA,Q,StatePoints,Game,State);
% else
%     Choice=QPolicy(ChoiceSet,SBSA,Q,StatePoints,Game,State);
% end
% %Where is the choice value in the lineup?
% ChosenVal=find(Choice==ChoiceSet);

%Greedy
    ChosenVal=find(ChoiceSet(:,2)==max(ChoiceSet(:,2)));

Move=ChosenVal(1);
CurMov=AllowedMoves(:,:,Move);


TPMNo = find(arrayfun(@(x) isequal(TPM(:,:,x),CurMov),1:size(TPM,3)));

TPMNoSupdate=TPMNo;

end



QPolicy
function Choice=QPolicy(ChoiceSet,SBSA,Q,StatePoints,Game,State)

    if Game==1
        Epi=0;
    else
        Epi=.5; %Train.5 Test 0
    end
[rowCS,ColCS]=size(ChoiceSet);
[rowQ,ColQ]=size(Q);
% disp('Next')

SpolGroup=[];


%     Qcol=Q(:,qq);
    QNA=Q(ChoiceSet(:,4:end));

    [AnomVal,anomloc]=max(sum(QNA,2));
    nomState=ChoiceSet(anomloc,4:end);

    %Policy Dev
    anom=nomState;
    SPolSet=[];
        for ii=1:1:rowCS

            a=ChoiceSet(ii,4:end);

            if a==anom
                SPol=1-Epi+(Epi/rowCS);
```

```
        else
            SPol=(Epi/rowCS);

        end
        SPolSet=[SPolSet;SPol];

    end
    SpolGroup=[SpolGroup,SPolSet];


Movegroup=(ChoiceSet(:,1)).';
SpolGroup=SpolGroup.';
p = cumsum([0; SpolGroup(1:end-1).'; 1+1e3*eps]);
[a a] = histc(rand,p);

Choice=Movegroup(a);
```

QPolicy2

```
function Choice=QPolicy2(ChoiceSet,SBSA,Q,StatePoints,Game,State)

  if Game==1
     Epi=0;
  else
     Epi=.5; %Train.5 Test 0
  end
[rowCS,ColCS]=size(ChoiceSet);
[rowQ,ColQ]=size(Q);
% disp('Next')
if ColQ~=1
  State=State(:).';
end
SpolGroup=[];

if Game==1
  SAPSet=[];
  for cs=1:length(ChoiceSet(:,1))
     SAP=[State,ChoiceSet(cs,1)];
     SAPSet=[SAPSet;SAP];

  end

  SAPSet;

SAPS="";

for i = 1:size(SAPSet, 1)
  SAPS=SAPS+"[";
  for j = 1:size(SAPSet, 2)
     if j == size(SAPSet, 2)
        SAPS=SAPS+sprintf("%1.0f", SAPSet(i, j));
     else
        SAPS=SAPS+sprintf("%1.0f,", SAPSet(i, j));
     end
  end
  if i == size(SAPSet, 1)
```

```
      SAPS=SAPS+"]";
   else
      SAPS=SAPS+"],";
   end
end

SAPS="["+SAPS+"]";

SAPS;

%Write to a python file for the functional approximator to evaluate
  [~,Qout]=system("python DLQOut.py "+SAPS);

  %Compare the FA Q value outputs to determine next action
  Qout=str2num(Qout(1048:end));
  QFA=[Qout,ChoiceSet(:,1)];
  [~,QUse]=max(QFA(:,1));
  Choice=QFA(QUse,2);



else

   %Determine the Q values based on the state action pairs
   QNA=Q(ChoiceSet(:,4:end));
   [AnomVal,anomloc]=max(sum(QNA,2));
   nomState=ChoiceSet(anomloc,4:end);

   %Policy Dev
   anom=nomState;
   SPolSet=[];
      for ii=1:1:rowCS

         a=ChoiceSet(ii,4:end);

         if a==anom
            SPol=1-Epi+(Epi/rowCS);

         else
            SPol=(Epi/rowCS);

         end
         SPolSet=[SPolSet;SPol];

      end
      SpolGroup=[SpolGroup,SPolSet];

%Choose the Next move based on the Q Values
Movegroup=(ChoiceSet(:,1)).';
SpolGroup=SpolGroup.';
p = cumsum([0; SpolGroup(1:end-1).'; 1+1e3*eps]);
[a a] = histc(rand,p);

Choice=Movegroup(a);

end
```

Random

```
function [Move,TPMNo]=random(AllowedMoves,TPM)

%Determine a move based on the remaining moves
[R,C,M]=size(AllowedMoves);
RandVal=randperm(M);
Move=RandVal(1);
CurMov=AllowedMoves(:,:,Move);
TPMNo = find(arrayfun(@(x) isequal(TPM(:,:,x),CurMov),1:size(TPM,3)));

end
```

isBox

```
function [YIB,WhichB]=isBox(UsedNodes,UsedNN,TPMNo,Boxes,Size,BoxWalls)

   YIB=[];
   WhichB=[];

   %Find which boxes were affect by the drawn line
   [x,y]=find(TPMNo==Boxes);
   BoxWalls=BoxWalls(x)+1;


   %Determine if and which box was drawn by seeing how many remaning lines need
   %to be drawn per box
   for ii=1:1:length(x)
      if ismember(Boxes(x(ii),1),UsedNN)==1 && ismember(Boxes(x(ii),2),UsedNN)==1 && ismember(Boxes(x(ii),3),UsedNN)==1
&& ismember(Boxes(x(ii),4),UsedNN)==1

         Which=[[Boxes(x(ii),1),Boxes(x(ii),2),Boxes(x(ii),3),Boxes(x(ii),4),Boxes(x(ii),1)]];
         WhichB=[WhichB;Which];
         YIB=[YIB,1];
      else
         Which=[0 0 0 0 0];
         WhichB=[WhichB;Which];
         YIB=[YIB,0];
      end
   end
   return
end
```

getSate

```
function [State,Stateset]=getState(UsedNN,SBSA,QuarterSets)


State=zeros(size(QuarterSets));
StateUpdate=find(ismember(QuarterSets,UsedNN));
State(StateUpdate)=1;
Stateset=[];
[StateSetsNo,~]=size(State);
   for qq=1:StateSetsNo
      [bool,cSL]=ismember(State(qq,:),SBSA,'rows');
      Stateset=[Stateset;cSL];
   end
end
```

writeoutDL

```
function Lernt=writeoutDL(DLSA,DLQEpisode,Size)

%This file writes the State Action paris and Q Values of the Episodes to a
%python file

someStateActions = DLSA;
someQs = DLQEpisode;

fid = fopen('States.py', 'w');
fprintf(fid, "import numpy as np \n");
fprintf(fid, "someStateActions = np.array([");
for i = 1:size(someStateActions, 1)
   fprintf(fid, "[");
   for j = 1:size(someStateActions, 2)
      if j == size(someStateActions, 2)
         fprintf(fid, "%f", someStateActions(i, j));
      else
         fprintf(fid, "%f,", someStateActions(i, j));
      end
   end
   if i == size(someStateActions, 1)
      fprintf(fid, "]");
   else
      fprintf(fid, "],");
   end
end
fprintf(fid, "]) \n");
fprintf(fid, "someQs = np.array([");
for i = 1:size(someQs, 1)
   fprintf(fid, "[");
   fprintf(fid, "%f", someQs(i));
   if i == size(someQs, 1)
      fprintf(fid, "]");
   else
      fprintf(fid, "],");
   end
end
fprintf(fid, "]) \n");

fclose(fid);

if Size==[3,3]
   [response,Lernt]=system("python DLQApprox3x3.py");

else
   [response,Lernt]=system("python DLQApprox.py");
end

end
```

Python

DLQApprox

```
import tensorflow
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import numpy as np
```

```
import States

#Check to see if the States imported are correct
print(States.someStateActions)

def Approximator(LStates):

    #Create the number of Layers used in FA
    Neurons1 = LStates
    Neurons2 = int(Neurons1/2)
    Neurons3 = int(Neurons2/2)

    # Build the Functional Approximator using the Neuron Layers
    FA = Sequential()
    FA.add(Dense(Neurons1, input_dim=LStates, kernel_initializer="normal", activation="relu"))
    FA.add(Dense(Neurons2, kernel_initializer="normal", activation="relu"))
    FA.add(Dense(Neurons3, kernel_initializer="normal", activation="relu"))


    # Output 1 value for the Q value when called
    FA.add(Dense(1, kernel_initializer="normal"))

    # Output the Mean Squared Error as a loss function for accurancy of FA
    FA.compile(loss="mean_squared_error", optimizer='adam')
    return FA



if __name__ == "__main__":

    #Check if Libraries load correctly
    print("Libraries Loaded Properly")

    #Pull the State Action Pairs from the python file written by Matlab
    inputStates = States.someStateActions

    #Pull the Q Tables from the python file written by Matlab
    labels=States.someQs

    #Create the variables in python from the imports
    PredictIPS = inputStates[-1][:]
    PredictLabels = labels[-1]

    #Get the Length of the States
    LStates=len(inputStates[0])


    #Build the Approximator
    FA = Approximator(LStates)
    FA.fit(inputStates, labels, epochs = 100, batch_size=len(inputStates))

    #Adjust the Shape so that Matlab can pull the Values
    Learnt = FA.predict(PredictIPS.reshape(13,1).T)

    #Output the Model and Labels to the terminal so Matlab can read it in
    print(Learnt)
    print(PredictLabels)

    #Save the Functional Approximator
```

```
    FA.save('DLQModel.h5')



DLQApprox3x3
import tensorflow
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import numpy as np
import States

#Check to see if the States imported are correct
print(States.someStateActions)

def Approximator(LStates):

    #Create the number of Layers used in FA
    Neurons1 = LStates
    Neurons2 = int(Neurons1/2)
    Neurons3 = int(Neurons2/2)

    # Build the Functional Approximator using the Neuron Layers
    FA = Sequential()
    FA.add(Dense(Neurons1, input_dim=LStates, kernel_initializer="normal", activation="relu"))
    FA.add(Dense(Neurons2, kernel_initializer="normal", activation="relu"))
    FA.add(Dense(Neurons3, kernel_initializer="normal", activation="relu"))


    # Output 1 value for the Q value when called
    FA.add(Dense(1, kernel_initializer="normal"))

    # Output the Mean Squared Error as a loss function for accurancy of FA
    FA.compile(loss="mean_squared_error", optimizer='adam')
    return FA



if __name__ == "__main__":

    #Check if Libraries load correctly
    print("Libraries Loaded Properly")

    #Pull the State Action Pairs from the python file written by Matlab
    inputStates = States.someStateActions

    #Pull the Q Tables from the python file written by Matlab
    labels=States.someQs

    #Create the variables in python from the imports
    PredictIPS = inputStates[-1][:]
    PredictLabels = labels[-1]

    #Get the Length of the States
    LStates=len(inputStates[0])


    #Build the Approximator
    FA = Approximator(LStates)
```

```
    FA.fit(inputStates, labels, epochs = 100, batch_size=len(inputStates))

    #Adjust the Shape so that Matlab can pull the Values
    Learnt = FA.predict(PredictIPS.reshape(49,1).T)

    #Output the Model and Labels to the terminal so Matlab can read it in
    print(Learnt)
    print(PredictLabels)

    #Save the Functional Approximator
    FA.save('DLQModel.h5')



DLQOut
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.models import load_model
import numpy as np
import sys


if __name__ == "__main__":

    #Load Model from Approximator
    DLQM=load_model('DLQModel.h5')

    #import the Matlab string from the terminal
    inputSAPS=str(sys.argv[1])

    #Evaluate the string as a numeric
    inputSAPS=eval(inputSAPS)

    #Make the numeric into an array
    inputSAPSL=np.array(inputSAPS)

    #Use the model from the approximator to predict the Q values given the state action pair
    Learnt = DLQM.predict(inputSAPSL)
    print(Learnt)
```