Training an Agent to Play Tic-Tac-Toe with Machine Learning

Jeremy Kritz 163006186

Final Project for Energy-Efficient Machine Learning

Project Type: Unspecified whether Type A or B

Prof Bo Yuan

5/16/2019

Abstract

The ability to learn to play games effectively has been a benchmark for artificial intelligence for decades. In recent years, machine learning, rather than hardcoded algorithms has proven to be an incredibly powerful tool in developing agents that can win at many different games. Tic-tac-toe is a very simple game compared to most, as it is solved, and has a relatively small state space. This project created a machine learning model that predicts a value of how good a move is for any given board state in a game of tic-tac-toe. The result is an agent that makes plays purely from machine learning, rather than a hardcoded AI, and wins significantly more against an imperfect, hardcoded AI than an agent that makes random plays.

Introduction

Reinforcement learning for games is one of the most interesting and cutting edge sectors of machine learning. For my final project I decided to focus on an incredibly simple game, tic-tac-toe.

I have been interested in machine learning related to game AI for a long time, and tic-tac-toe felt like a good place to start. For my final project, I created an agent to play tic-tac-toe that relies exclusively on machine learning for its decision making. The model was trained on tens of thousands of games of tic-tac-toe, and the result was an agent that can beat an imperfect hardcoded AI more than five times more often than an agent that makes only random plays. The result is inferior to a hardcoded agent, but as tic-tac-toe is a solved game, beating a perfectly hardcoded agent is not possible.

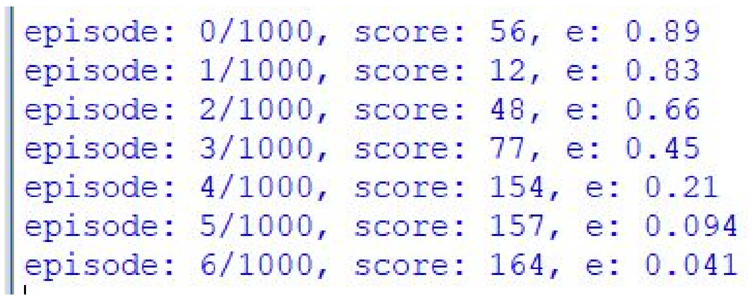
The demonstration starts on page 15.

Related Work

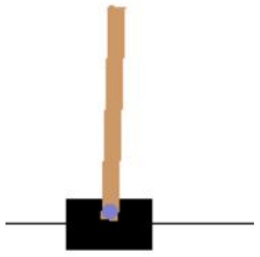
Google's DeepMind team recently created models that beat experts at Go and Starcraft 2, games which until recently were thought near impossible for AI to effectively play due to the nearly infinite number of potential states [1].

To test out reinforcement learning, I implemented a tutorial to play CartPole, a game where a cart is controlled to keep a pole from falling for as long as possible [2]. There is a library for reinforcement learning called Gym, which was used in this demo, although it was too complicated to be used in the final project. Nevertheless, this mini-project was useful in demonstrating the basics of how a game-playing agent can be trained without a starting data set. I only ran a few games of CartPole, but the agent showed clear, fast improvement.

[1] https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/
[2] https://keon.io/deep-q-learning/



```
episode: 0/1000, score: 56, e: 0.89
episode: 1/1000, score: 12, e: 0.83
episode: 2/1000, score: 48, e: 0.66
episode: 3/1000, score: 77, e: 0.45
episode: 4/1000, score: 154, e: 0.21
episode: 5/1000, score: 157, e: 0.094
episode: 6/1000, score: 164, e: 0.041
```



Data Description

Because this project is about AI, I did not want to get too into the weeds with creating the tic-tac-toe game. The methods to create the game board, the hardcoded AI, and basic methods such as checking is a space is empty, and making a random move, were lightly adapted from an online tutorial [3].

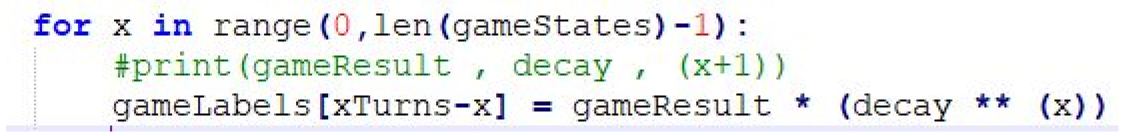
One of the biggest things separating this problem from the projects tackled in class is the lack of an existing dataset. I had to create a dataset of game states, and their label, which in this case was a reward function.

Something to note here is that my implementation would probably be called pseudo-reinforcement learning. The training set came entirely from a hardcoded AI playing itself, and not from the agent playing the game.

I had to come up with a reward function each move x made in each game. My reward function was adapted from the reward function used in a Deep-Q Network. This is explored in DeepMind's Atari project [4].

[3] https://inventwithpython.com/chapter10.html
[4] https://deepmind.com/research/publications/playing-atari-deep-reinforcement-learning/

$$loss = \left(\overset{\text{Reward}}{r} + \overset{\text{Decay Rate}}{\gamma} \underbrace{\max_{a`} \hat{Q}(s, a`)}_{\text{Target}} - \underbrace{Q(s, a)}_{\text{Prediction}} \right)^2$$

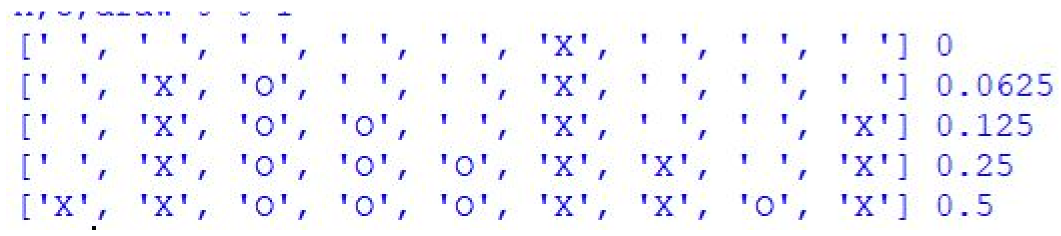

```
for x in range(0,len(gameStates)-1):
    #print(gameResult , decay , (x+1))
    gameLabels[xTurns-x] = gameResult * (decay ** (x))
```

TurnValue = GameResult * Decay^TurnsSinceEndOfGame

Each game state was given a label by the equation above. A losing board would have a negative value, a board a turn away from losing would have a smaller negative value, a turn 2 turns away from losing would have a smaller negative value, and so on. The same is true for winning, but with a positive value.

The initial model training used rewards of -0.95 for losing, 0.2 for a draw, and 0.95 for a victory. These were arbitrary.

The draw reward was changed to 0.5 for further training, when it was discovered that the agent never drew a game, while in perfect tic-tac-toe play, a draw is often the best outcome.

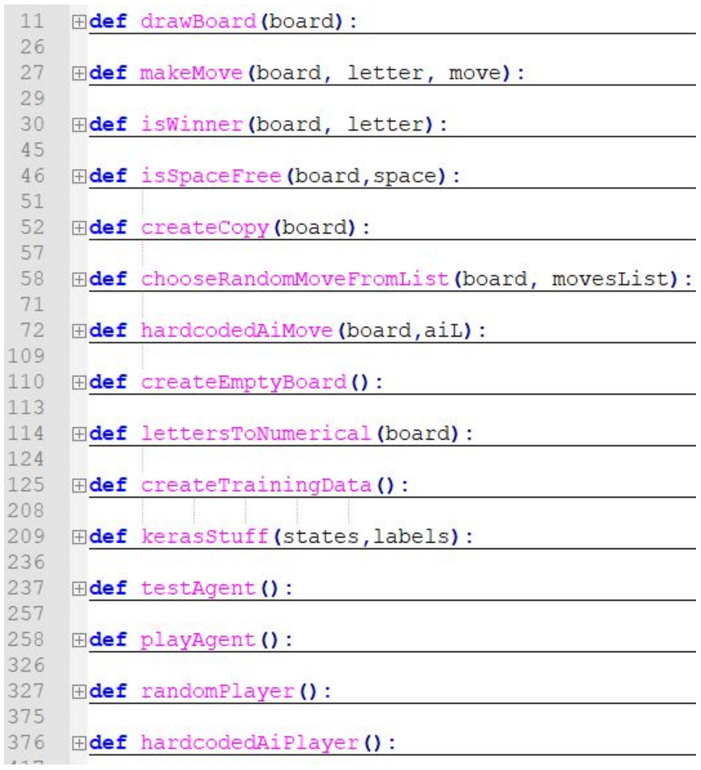

```
['  ', ' ', ' ', ' ', ' ', 'X', ' ', ' ', ' '] 0
[' ', 'X', 'O', ' ', ' ', 'X', ' ', ' ', ' '] 0.0625
[' ', 'X', 'O', 'O', ' ', 'X', ' ', ' ', 'X'] 0.125
[' ', 'X', 'O', 'O', 'O', 'X', 'X', ' ', 'X'] 0.25
['X', 'X', 'O', 'O', 'O', 'X', 'X', 'O', 'X'] 0.5
```

Training data is saved as shown above. The 5 states above show a game that resulted in a draw, and the rewards associated with each turn.

The characters are changed to float32 values at time of play, and at time of training.

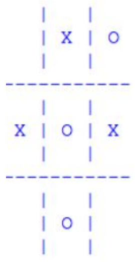
Method Description

There are 15 methods, a number of which are trivial.



```
def drawBoard(board):
def makeMove(board, letter, move):
def isWinner(board, letter):
def isSpaceFree(board,space):
def createCopy(board):
def chooseRandomMoveFromList(board, movesList):
def hardcodedAiMove(board,aiL):
def createEmptyBoard():
def lettersToNumerical(board):
def createTrainingData():
def kerasStuff(states,labels):
def testAgent():
def playAgent():
def randomPlayer():
def hardcodedAiPlayer():
```

Draw board() takes a 9-length array of ' ','O' and 'X' and returns a printed representation of that game state.



```
   |   |
   | X | O
   |   |
-----------
   |   |
 X | O | X
   |   |
-----------
   |   |
   | O |
   |   |
```

makeMove() changes the inputted array to include the inputted letter, and the inputted spot.

The move numbers correspond to the following board spaces.

0 1 2

3 4 5

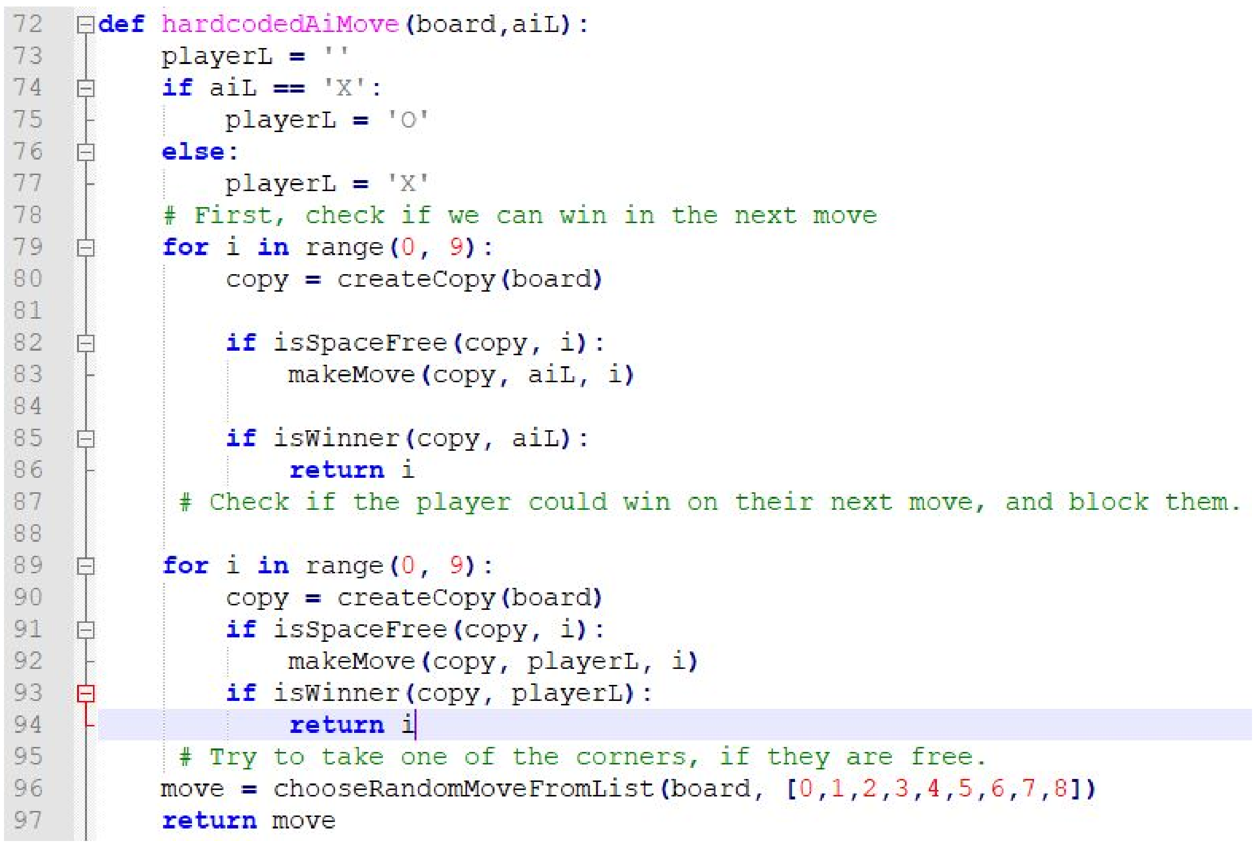
6 7 8

isWinner() checks if a game state is a winning position for the inputted letter.

isSpaceFree() seems self-explanatory. It returns a boolean.
createCopy() returns a copy of the inputted board, this is useful for checking the results of moves before actually making those moves on the main board.

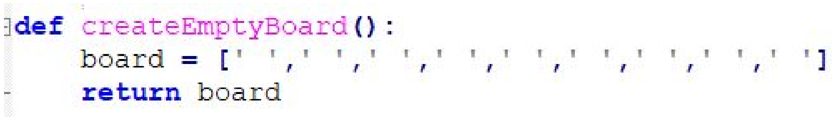
chooseRandomMoveFromList() chooses a random number from the inputted array, and returns that integer if that space is free on the inputted board. If the space is not free, the array element is discarded and the process repeats. If no elements are free, or the list is empty, the method returns None.

hardcodedAiMove() this method returns the integer corresponding to a winning move if possible, if not it returns a move to stop the other player from winning, and if neither is an option, returns a random move. If there are no valid moves, the method returns None. The method this was adapted from was a stronger AI, but I made it deliberately weaker to allow more variety of games, as well as to allow the trained agent to be able to win.



```
def hardcodedAiMove(board,aiL):
    playerL = ''
    if aiL == 'X':
        playerL = 'O'
    else:
        playerL = 'X'
    # First, check if we can win in the next move
    for i in range(0, 9):
        copy = createCopy(board)

        if isSpaceFree(copy, i):
            makeMove(copy, aiL, i)

        if isWinner(copy, aiL):
            return i
     # Check if the player could win on their next move, and block them.

    for i in range(0, 9):
        copy = createCopy(board)
        if isSpaceFree(copy, i):
            makeMove(copy, playerL, i)
        if isWinner(copy, playerL):
            return i
     # Try to take one of the corners, if they are free.
    move = chooseRandomMoveFromList(board, [0,1,2,3,4,5,6,7,8])
    return move
```

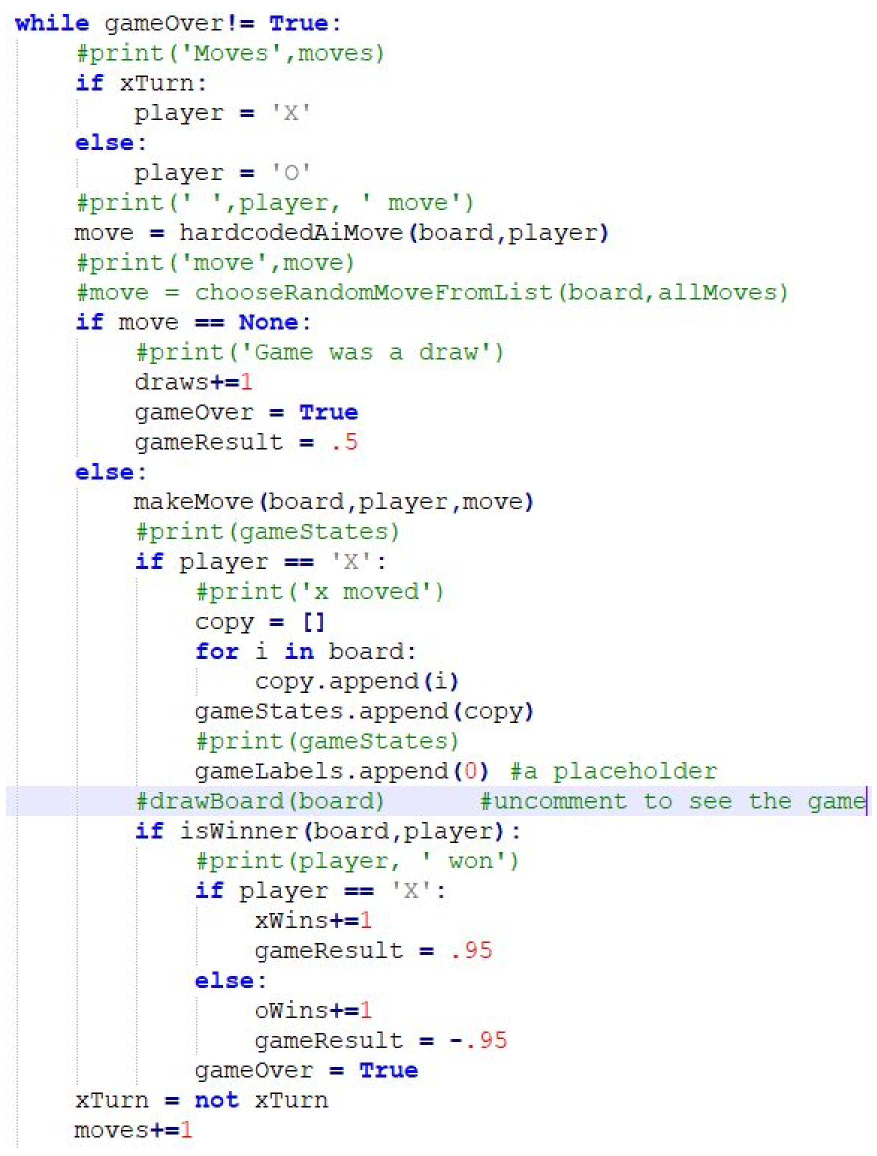
createEmptyBoard()



```
def createEmptyBoard():
    board = [' ',' ',' ',' ',' ',' ',' ',' ',' ']
    return board
```

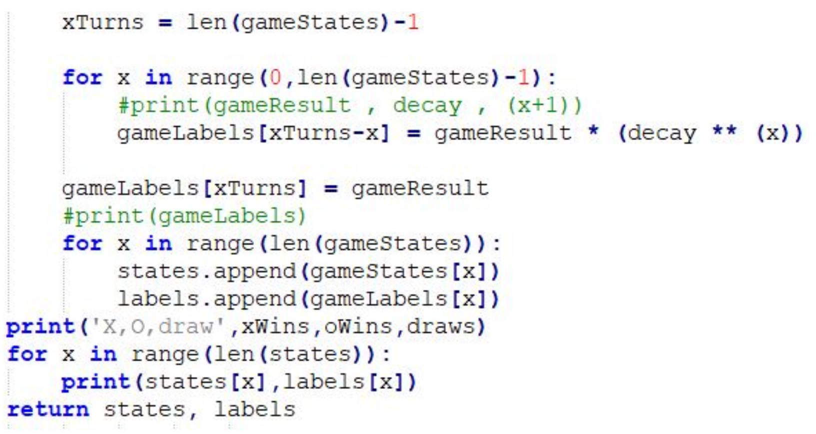
lettersToNumerical() this method takes an inputted array of characters, and returns an array of numbers, 'X' is set equal to 0.5, 'O' is set equal to -0.5, and set blank spaces to 0.0.

createTrainingData() plays a specified number of games, with both player making moves with the hardcoded AI method described above. The part of the method that plays each game is shown below.



```
while gameOver!= True:
    #print('Moves',moves)
    if xTurn:
        player = 'X'
    else:
        player = 'O'
    #print(' ',player, ' move')
    move = hardcodedAiMove(board,player)
    #print('move',move)
    #move = chooseRandomMoveFromList(board,allMoves)
    if move == None:
        #print('Game was a draw')
        draws+=1
        gameOver = True
        gameResult = .5
    else:
        makeMove(board,player,move)
        #print(gameStates)
        if player == 'X':
            #print('x moved')
            copy = []
            for i in board:
                copy.append(i)
            gameStates.append(copy)
            #print(gameStates)
            gameLabels.append(0)  #a placeholder
        #drawBoard(board)      #uncomment to see the game
        if isWinner(board,player):
            #print(player, ' won')
            if player == 'X':
                xWins+=1
                gameResult = .95
            else:
                oWins+=1
                gameResult = -.95
            gameOver = True
    xTurn = not xTurn
    moves+=1
```

After each game, the labels corresponding to each move by X are generated, then all states and corresponding labels are added to arrays containing data for all games, as shown below.



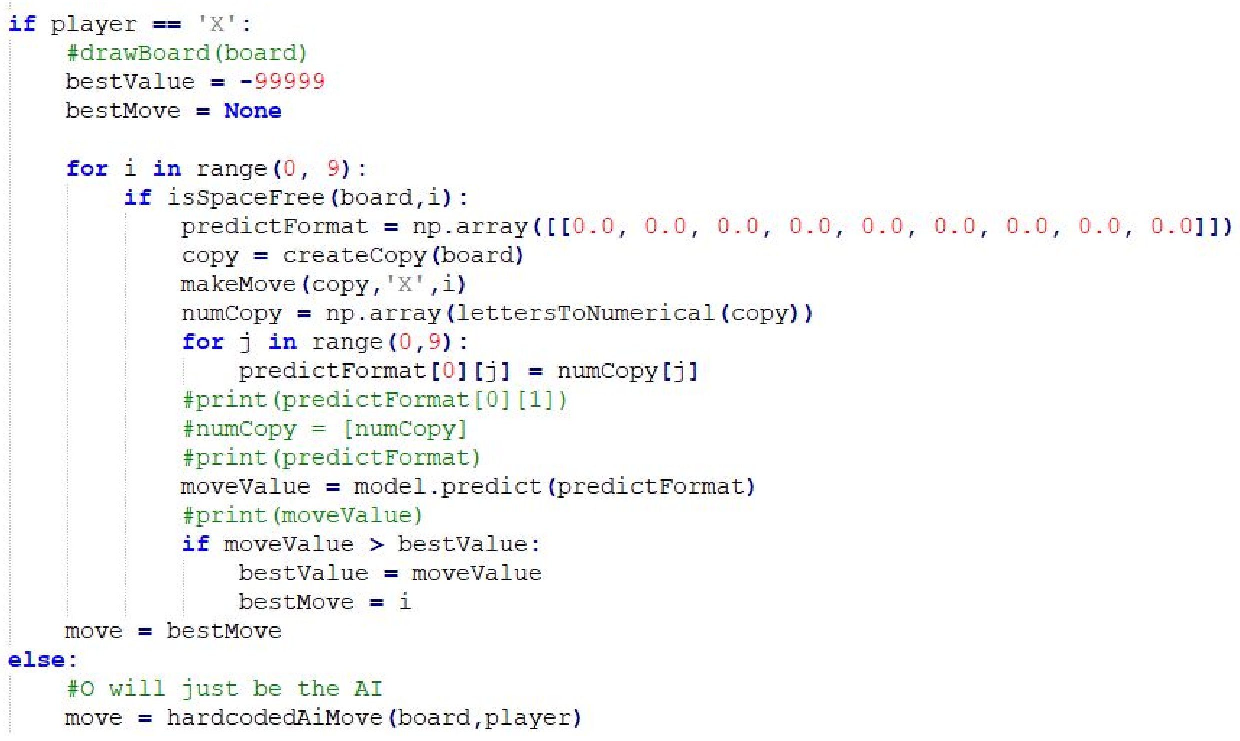
```
    xTurns = len(gameStates)-1

    for x in range(0,len(gameStates)-1):
        #print(gameResult , decay , (x+1))
        gameLabels[xTurns-x] = gameResult * (decay ** (x))

    gameLabels[xTurns] = gameResult
    #print(gameLabels)
    for x in range(len(gameStates)):
        states.append(gameStates[x])
        labels.append(gameLabels[x])
print('X,O,draw',xWins,oWins,draws)
for x in range(len(states)):
    print(states[x],labels[x])
return states, labels
```

In training, the method was made to create 10,000 games, resulting in around 44,000 states, although it is likely these are not all unique states.

kerasStuff() this method contains the model, builds it, trains it, and saves. After initial training, the model's creation was commented out, and replaced with a line to load the model to train further. The model is described in more detail below, in the next section.

testAgent() this method is not important to the final result or testing, and was used to find initial success, as shown in the first figure in the Results section.

playAgent() this method loads the model, and plays a specified number of games. X's moves are made by predicting the reward for each possible next state, and choosing the best result. O's moves are made by hardcodedAiMove(). It keeps track of the result of each game.
These moves are shown below.



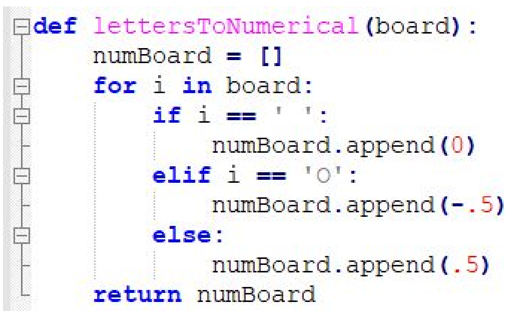
```
if player == 'X':
    #drawBoard(board)
    bestValue = -99999
    bestMove = None

    for i in range(0, 9):
        if isSpaceFree(board,i):
            predictFormat = np.array([[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]])
            copy = createCopy(board)
            makeMove(copy,'X',i)
            numCopy = np.array(lettersToNumerical(copy))
            for j in range(0,9):
                predictFormat[0][j] = numCopy[j]
            #print(predictFormat[0][1])
            #numCopy = [numCopy]
            #print(predictFormat)
            moveValue = model.predict(predictFormat)
            #print(moveValue)
            if moveValue > bestValue:
                bestValue = moveValue
                bestMove = i
    move = bestMove
else:
    #O will just be the AI
    move = hardcodedAiMove(board,player)
```

randomPlayer() this method is essentially identical to playAgent(), except X's move is made by chooseRandomMoveFromList([0,1,2,3,4,5,6,7,8])

hardcodedAiPlayer() this method is essentially identical to playAgent() and randomPlayer(), except X's moves are made by hardcodedAiMove()

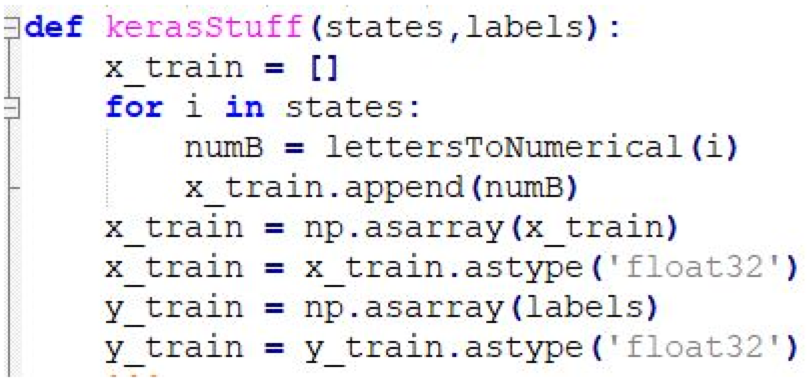
Model Description

For the model, I used Keras. At first, I thought that the output would be a matrix of length 9, with a softmax activation, to rate the best possible move. I then realized it was simpler to have a single output, to estimate the expected reward of a given state.
The first issue was converting the 3 possible states (' ', 'X', 'O') to numerical values that the model could understand. I created a method lettersToNumerical(), that would make X equal to 0.5, O equal to -0.5, and set blank spaces to 0.



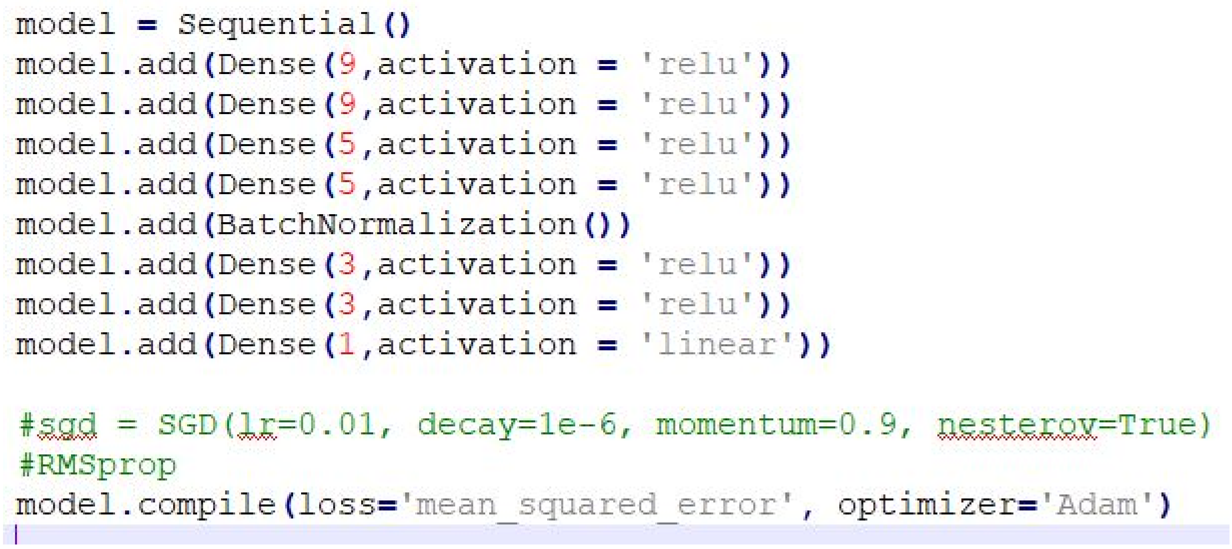
```
def lettersToNumerical(board):
    numBoard = []
    for i in board:
        if i == ' ':
            numBoard.append(0)
        elif i == 'O':
            numBoard.append(-.5)
        else:
            numBoard.append(.5)
    return numBoard
```

Training data is read into the model as follows.



```
def kerasStuff(states,labels):
    x_train = []
    for i in states:
        numB = lettersToNumerical(i)
        x_train.append(numB)
    x_train = np.asarray(x_train)
    x_train = x_train.astype('float32')
    y_train = np.asarray(labels)
    y_train = y_train.astype('float32')
```

The structure of the model is as follows.

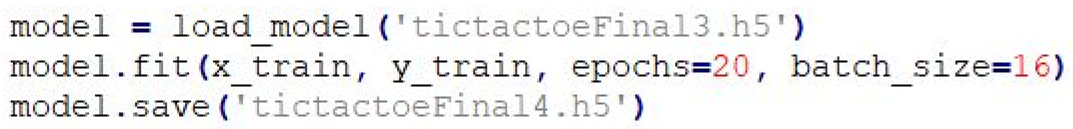


```
model = Sequential()
model.add(Dense(9,activation = 'relu'))
model.add(Dense(9,activation = 'relu'))
model.add(Dense(5,activation = 'relu'))
model.add(Dense(5,activation = 'relu'))
model.add(BatchNormalization())
model.add(Dense(3,activation = 'relu'))
model.add(Dense(3,activation = 'relu'))
model.add(Dense(1,activation = 'linear'))

#sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
#RMSprop
model.compile(loss='mean_squared_error', optimizer='Adam')
```

I decided to only use dense (fully convolutional) layers, as the 9-length input seemed small for a Conv2D layer, with stride and filters. Not only that, but the idea of regions as detected by a filter for an image is not the same as it would be for a tic-tac-toe board.

In theory, I could have created layers larger than 9 to make a larger model, but I have only ever seen models with a "funnel" shape, continuously getting smaller, and I used that approach here. The structure, 2 layers each of sizes 9, 5 and 3 were chosen arbitrarily. Finally, I had a single output with linear activation, as it was estimating a score, not making a prediction. An early version used softmax, which did not work.

In future training, training was done by loading the model rather than building it.



```
model = load_model('tictactoeFinal3.h5')
model.fit(x_train, y_train, epochs=20, batch_size=16)
model.save('tictactoeFinal4.h5')
```

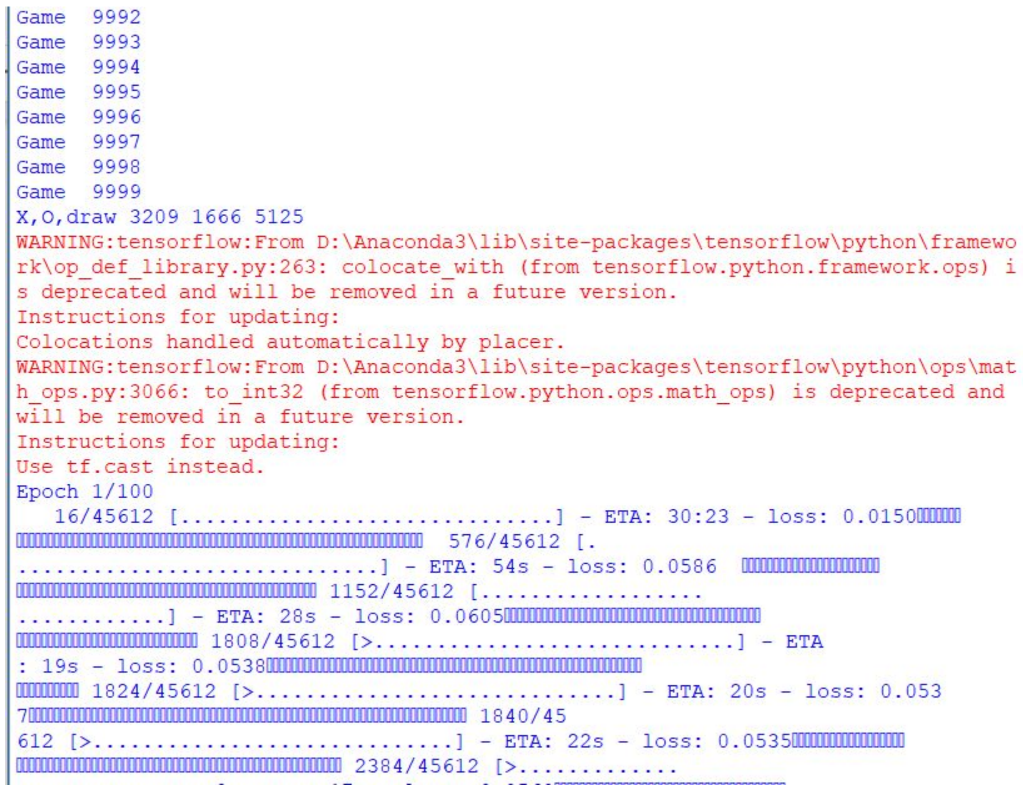
Experimental Procedure

The way the agent plays is simple. The model takes a game state, and returns a score. The more optimal the position for X, the higher the score. The agent tests the states that would result from all valid plays, and makes the move that results in the highest scoring state.

To test the model, I had it play against a hardcoded AI. This AI was not an optimal AI (tic-tac-toe is a solved game as xkcd's map helpfully demonstrates [5]) because I wanted the ML agent to be able to win, and a perfect tic-tac-toe algorithm is unbeatable. As described above in the Method Description section, the AI made a winning move if possible, if not it made a move to stop the other player from winning, and if neither was an option, made a random move.

To test the agent, it played 200 games as X, and recorded all wins, losses, and draws. As a control group, I also ran 200 games with X making random moves, and O as the hardcoded imperfect AI described above. With more and more training, the model showed marked improvement over random play, proving that the model was effectively learning how to play tic-tac-toe.

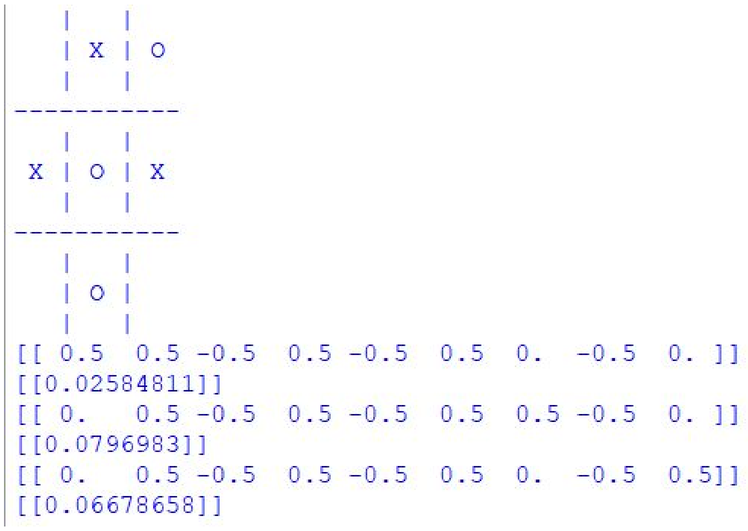
[5] https://xkcd.com/832/



```
Game  9992
Game  9993
Game  9994
Game  9995
Game  9996
Game  9997
Game  9998
Game  9999
X,O,draw 3209 1666 5125
WARNING:tensorflow:From D:\Anaconda3\lib\site-packages\tensorflow\python\framewo
rk\op_def_library.py:263: colocate_with (from tensorflow.python.framework.ops) i
s deprecated and will be removed in a future version.
Instructions for updating:
Colocations handled automatically by placer.
WARNING:tensorflow:From D:\Anaconda3\lib\site-packages\tensorflow\python\ops\mat
h_ops.py:3066: to_int32 (from tensorflow.python.ops.math_ops) is deprecated and
will be removed in a future version.
Instructions for updating:
Use tf.cast instead.
Epoch 1/100
   16/45612 [..............................] - ETA: 30:23 - loss: 0.0150▯▯▯▯▯▯▯▯▯
▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯  576/45612 [.
.............................] - ETA: 54s - loss: 0.0586  ▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯
▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯ 1152/45612 [..................
...........] - ETA: 28s - loss: 0.0605▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯
▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯ 1808/45612 [>.............................] - ETA
: 19s - loss: 0.0538▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯
▯▯▯▯▯▯▯▯▯▯▯▯ 1824/45612 [>.............................] - ETA: 20s - loss: 0.053
7▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯ 1840/45
612 [>.............................] - ETA: 22s - loss: 0.0535▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯
▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯ 2384/45612 [>.............
```

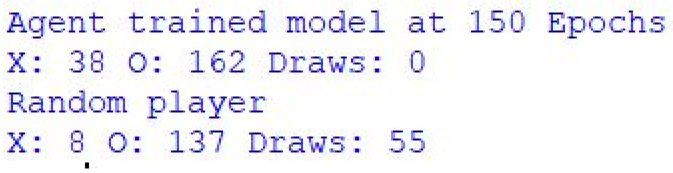
Results

First signs of success came when after around 100 epochs, the model could recognize a winning move as the best move. The test method evaluates moves, left to right, top to bottom, and the result can be seen below. The best move for X is to block O in the bottom left corner, the second move. This is the highest valued move, meaning the model correctly identified the correct play. This proof of concept led me to actually start getting real results by recording the outcomes of games.



```
   |   |
   | X | O
   |   |
-----------
   |   |
 X | O | X
   |   |
-----------
   |   |
   | O |
   |   |
[[ 0.5  0.5 -0.5  0.5 -0.5  0.5  0.  -0.5  0. ]]
[[0.02584811]]
[[ 0.   0.5 -0.5  0.5 -0.5  0.5  0.5 -0.5  0. ]]
[[0.0796983]]
[[ 0.   0.5 -0.5  0.5 -0.5  0.5  0.  -0.5  0.5]]
[[0.06678658]]
```

The model was initially trained on 10000 games, each game having 3 to 5 labelled moves by X. The model was trained for 100 epochs in this data set with a batch size of 16, then 50 more epochs with a batch size of 8. Each new training session used a new training set of 10000 games. There are over 200,000 possible games of tic-tac-toe, so I decided to keep remaking the dataset to try to find more states.

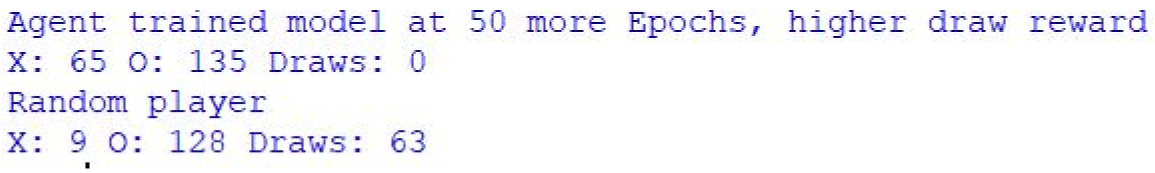


```
Agent trained model at 150 Epochs
X: 38 O: 162 Draws: 0
Random player
X: 8 O: 137 Draws: 55
```

This was a very interesting result. The agent won 4.75 times more often than random plays, but also lost more often, as it was avoiding draws for some reason.

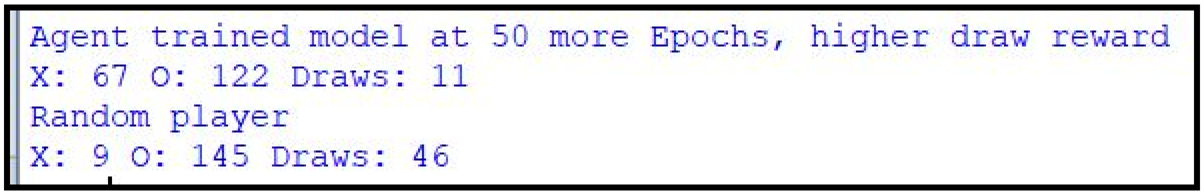
To fix this, I decided to train the model more (not start over) using new training data. In all future training after this, the end reward for a drawn game was 0.5, up from 0.2 in the initial training.

After 50 epochs with a batch size of 16, and a new training set, the model still did not draw, but won much more.



```
Agent trained model at 50 more Epochs, higher draw reward
X: 65 O: 135 Draws: 0
Random player
X: 9 O: 128 Draws: 63
```

Another 50 Epochs finally yielded draws. For the first time, the model not only won far more than the random player, but also lost less often.



```
Agent trained model at 50 more Epochs, higher draw reward
X: 67 O: 122 Draws: 11
Random player
X: 9 O: 145 Draws: 46
```

Continued training after this led to diminishing returns and overfitting. A larger model might be able to train to higher win rates.

The final model trained for 250 Epochs on datasets of 10,000 games, although the games within these sets varied throughout training.

This result, however meant winning more than 6 times more often than a random player, proving that the agent had learned effectively.

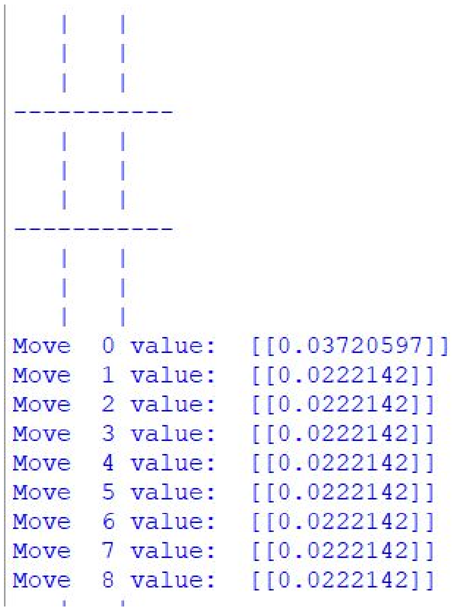
Let's go through a game where the agent wins.

Remember what moves correspond to what board spaces

0 1 2

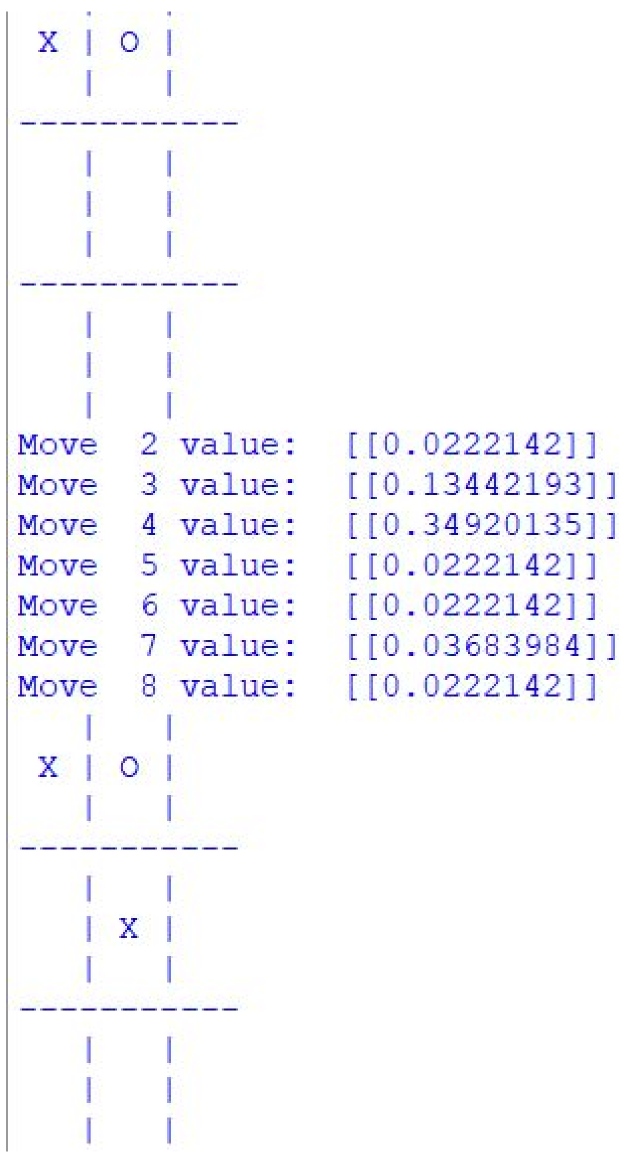
3 4 5

6 7 8



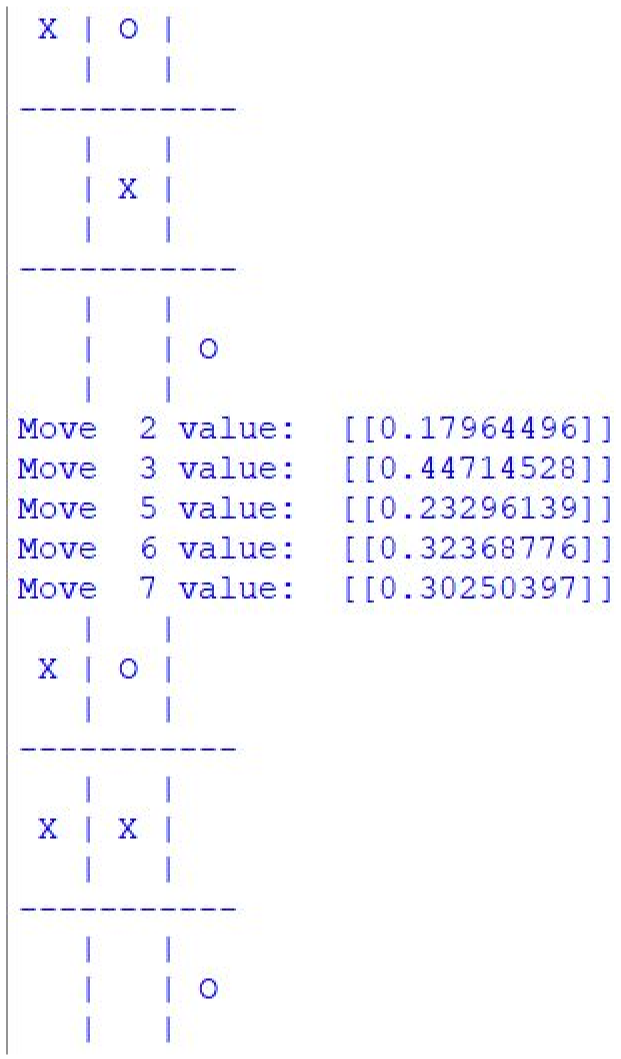
```
   |   |
   |   |
   |   |
-----------
   |   |
   |   |
   |   |
-----------
   |   |
   |   |
   |   |
Move  0 value:  [[0.03720597]]
Move  1 value:  [[0.0222142]]
Move  2 value:  [[0.0222142]]
Move  3 value:  [[0.0222142]]
Move  4 value:  [[0.0222142]]
Move  5 value:  [[0.0222142]]
Move  6 value:  [[0.0222142]]
Move  7 value:  [[0.0222142]]
Move  8 value:  [[0.0222142]]
```

The agent has learned to start in the corner. This is optimal play.



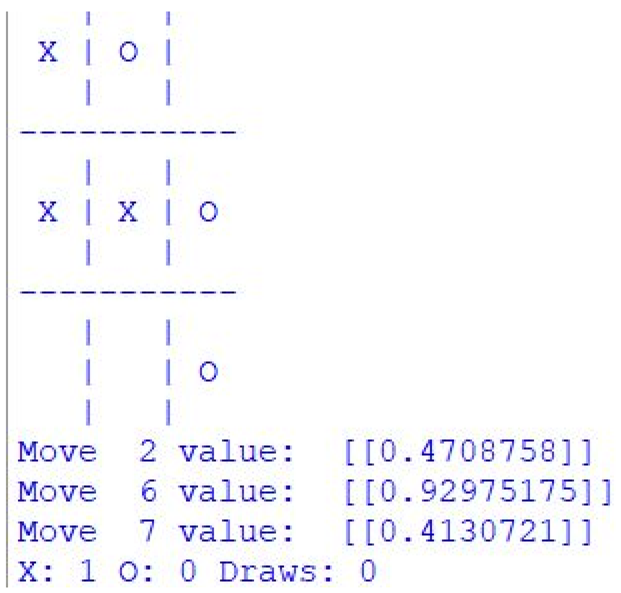
```
 X | O |
   |   |
-----------
   |   |
   |   |
   |   |
-----------
   |   |
   |   |
   |   |
Move  2 value:  [[0.0222142]]
Move  3 value:  [[0.13442193]]
Move  4 value:  [[0.34920135]]
Move  5 value:  [[0.0222142]]
Move  6 value:  [[0.0222142]]
Move  7 value:  [[0.03683984]]
Move  8 value:  [[0.0222142]]
   |   |
 X | O |
   |   |
-----------
   |   |
   | X |
   |   |
-----------
   |   |
   |   |
   |   |
```

The model ranks space 4 (the center) as the highest rated move. Remember, it is predicting based on the board resulting from making that play.



```
 X | O |
   |   |
-----------
   |   |
   | X |
   |   |
-----------
   |   |
   |   | O
   |   |
Move  2 value:  [[0.17964496]]
Move  3 value:  [[0.44714528]]
Move  5 value:  [[0.23296139]]
Move  6 value:  [[0.32368776]]
Move  7 value:  [[0.30250397]]
   |   |
 X | O |
   |   |
-----------
   |   |
 X | X |
   |   |
-----------
   |   |
   |   | O
   |   |
```

O moves to block. The agent correctly rates space 3 as highest. This sets up a victory on X's next turn, no matter where O plays. Remember that our training data gives its second highest reward value to boards 1 turn away from victory, such as this one.



```
   |   |
 X | O |
   |   |
-----------
   |   |
 X | X | O
   |   |
-----------
   |   |
   |   | O
   |   |
Move  2 value:  [[0.4708758]]
Move  6 value:  [[0.92975175]]
Move  7 value:  [[0.4130721]]
X: 1 O: 0 Draws: 0
```

The model gives a good score for blocking O from winning, and by far the best score to the winning board.

The board is not drawn here, but X makes the winning move (space 6), and the method notes that X has won.

Conclusion

This project shows that machine learning was effectively used to train an agent to play tic-tac-toe.

These results of course, were not perfect. This paper, and project serve as a proof of concept that machine learning was effectively used to improve an AI agents play, with no hardcoded algorithms. Even though the ML AI was in this case significantly worse than a hardcoded one, this is due to the fact that a hardcoded solution to tic-tac-toe is trivial compared to games with exponentially larger possible states. For games where a hardcoded solution is not possible, such as Go, machine learning approaches are the clear way forward.

In theory, the tic-tac-toe agent could be improved to play better in a few ways. This was not quite true reinforcement learning, in that the agent never used its own model to

train, only the states and rewards from matches with hardcoded players. The standard approach to reinforcement learning, having the agent with the model play and improve might yield a better result.
It is also likely that a significantly larger model would also result in a more accurate predictor, and thus a higher win rate. Finally, pretty much all values used were arbitrary. Values chosen to represent X and O, the decay rate, and the reward values were all arbitrary, and it is possible other values would be more optimal.

Acknowledgements

Thanks to professor Yuan. I would not have been able to complete this project before taking this class.

References

The following sources were used for research, along with all references in footnotes.

1. https://medium.com/@shayak_89588/playing-ultimate-tic-tac-toe-with-reinforcement-learning-7bea5b9d7252
2. https://becominghuman.ai/reinforcement-learning-step-by-step-17cde7dbc56c

Appendix

Source Code

CartPole

```
import random
import gym
import numpy as np
from collections import deque
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam

EPISODES = 1000

class DQNAgent:
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size
        self.memory = deque(maxlen=2000)
        self.gamma = 0.95    # discount rate
        self.epsilon = 1.0  # exploration rate
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.995
        self.learning_rate = 0.001
        self.model = self._build_model()

    def _build_model(self):
        # Neural Net for Deep-Q learning Model
        model = Sequential()
        model.add(Dense(24, input_dim=self.state_size, activation='relu'))
        model.add(Dense(24, activation='relu'))
        model.add(Dense(self.action_size, activation='linear'))
        model.compile(loss='mse',
                      optimizer=Adam(lr=self.learning_rate))
        return model

    def remember(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state, done))

    def act(self, state):
        if np.random.rand() <= self.epsilon:
            return random.randrange(self.action_size)
        act_values = self.model.predict(state)
        return np.argmax(act_values[0])  # returns action

    def replay(self, batch_size):
        minibatch = random.sample(self.memory, batch_size)
        for state, action, reward, next_state, done in minibatch:
            target = reward
            if not done:
                target = (reward + self.gamma *
                          np.amax(self.model.predict(next_state)[0]))
            target_f = self.model.predict(state)
            target_f[0][action] = target
            self.model.fit(state, target_f, epochs=1, verbose=0)
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay

    def load(self, name):
        self.model.load_weights(name)

    def save(self, name):
        self.model.save_weights(name)


if __name__ == "__main__":
    env = gym.make('CartPole-v1')
    state_size = env.observation_space.shape[0]
    action_size = env.action_space.n
    agent = DQNAgent(state_size, action_size)
    agent.load("cartpole-dqn.h5")
    done = False
    batch_size = 32

    for e in range(EPISODES):
        state = env.reset()
        state = np.reshape(state, [1, state_size])
        for time in range(500):
            env.render()
            action = agent.act(state)
            next_state, reward, done, _ = env.step(action)
            reward = reward if not done else -10
            next_state = np.reshape(next_state, [1, state_size])
        agent.remember(state, action, reward, next_state, done)
        state = next_state
        if done:
            print("episode: {}/{}, score: {}, e: {:.2}"
                  .format(e, EPISODES, time, agent.epsilon))
            break
        if len(agent.memory) > batch_size:
            agent.replay(batch_size)
    if e % 10 == 0:
        agent.save("cartpole-dqn.h5")
```

TicTacToe

```
import numpy as np
import random
import gym
from keras.models import Sequential
from keras.layers import Dense, BatchNormalization
from keras.optimizers import SGD, RMSprop,Adam
from collections import deque
from keras.models import load_model


def drawBoard(board):

# This function prints out the board that it was passed.

        print('   |   |')
        print(' ' + board[0] + ' | ' + board[1] + ' | ' + board[2])
        print('   |   |')
        print('-----------')
        print('   |   |')
        print(' ' + board[3] + ' | ' + board[4] + ' | ' + board[5])
        print('   |   |')
        print('-----------')
        print('   |   |')
        print(' ' + board[6] + ' | ' + board[7] + ' | ' + board[8])
        print('   |   |')

def makeMove(board, letter, move):
        board[move] = letter

def isWinner(board, letter):

# Given a board and a player’s letter, this function returns True if that player has won.


        return ((board[0] == letter and board[1] == letter and board[2] == letter) or # across the top
        (board[3] == letter and board[4] == letter and board[5] == letter) or # across the middle
        (board[6] == letter and board[7] == letter and board[8] == letter) or # across the board

        (board[0] == letter and board[3] == letter and board[6] == letter) or # down the letterft side
        (board[1] == letter and board[4] == letter and board[7] == letter) or # down the middle
        (board[2] == letter and board[5] == letter and board[8] == letter) or # down the right side

        (board[0] == letter and board[4] == letter and board[8] == letter) or # diagonal
        (board[2] == letter and board[4] == letter and board[6] == letter)) # diagonal

def isSpaceFree(board,space):
        if board[space] == '' or board[space] == ' ':
                return True
        else:
                return False

def createCopy(board):
        copy = []
```

```
	for i in board:
		copy.append(i)
	return copy

def chooseRandomMoveFromList(board, movesList):

	# Returns a valid move from the passed list on the passed board.

	# Returns None if there is no valid move.
	possibleMoves = []
	for i in movesList:
		if isSpaceFree(board, i):
			possibleMoves.append(i)
	if len(possibleMoves) != 0:
		return random.choice(possibleMoves)
	else:
		return None

def hardcodedAiMove(board,aiL):
	playerL = ''
	if aiL == 'X':
		playerL = 'O'
	else:
		playerL = 'X'
	# First, check if we can win in the next move
	for i in range(0, 9):
		copy = createCopy(board)

		if isSpaceFree(copy, i):
			makeMove(copy, aiL, i)

		if isWinner(copy, aiL):
			return i
	 # Check if the player could win on their next move, and block them.

	for i in range(0, 9):
		copy = createCopy(board)
		if isSpaceFree(copy, i):
			makeMove(copy, playerL, i)
		if isWinner(copy, playerL):
			return i
	 # Try to take one of the corners, if they are free.
	move = chooseRandomMoveFromList(board, [0,1,2,3,4,5,6,7,8])
	return move



	#making the ai a bit worse intentionally to add variety
	'''
	move = chooseRandomMoveFromList(board, [0, 2, 6, 8])
	if move!= None:
		return move
	#take center
	if isSpaceFree(board,4):
		return 4
	return chooseRandomMoveFromList(board, [1, 3, 5, 7])
	'''

def createEmptyBoard():
	board = [' ',' ',' ',' ',' ',' ',' ',' ',' ']
	return board

def lettersToNumerical(board):
	numBoard = []
	for i in board:
		if i == ' ':
```

```
                numBoard.append(0)
            elif i == 'O':
                numBoard.append(-.5)
            else:
                numBoard.append(.5)
    return numBoard

def createTrainingData():
    print('hi')
    decay = .5
    xWins = 0
    draws = 0
    oWins = 0
    states = []
    player = ''
    labels = []
    numGames = 10000
    allMoves = [0,1,2,3,4,5,6,7,8]
    for i in range(numGames):
        print('Game ',i)
        gameStates = []
        gameLabels = []
        board = createEmptyBoard()
        gameOver = False
        xTurn = True
        gameResult = 0
        moves = 0
        while gameOver!= True:
            #print('Moves',moves)
            if xTurn:
                player = 'X'
            else:
                player = 'O'
            #print(' ',player, ' move')
            move = hardcodedAiMove(board,player)
            #print('move',move)
            #move = chooseRandomMoveFromList(board,allMoves)
            if move == None:
                #print('Game was a draw')
                draws+=1
                gameOver = True
                gameResult = .5
            else:
                makeMove(board,player,move)
                #print(gameStates)
                if player == 'X':
                    #print('x moved')
                    copy = []
                    for i in board:
                        copy.append(i)
                    gameStates.append(copy)
                    #print(gameStates)
                    gameLabels.append(0) #a placeholder
                #drawBoard(board)     #uncomment to see the game
                if isWinner(board,player):
                    #print(player, ' won')
                    if player == 'X':
                        xWins+=1
                        gameResult = .95
                    else:
                        oWins+=1
                        gameResult = -.95
                    gameOver = True
            xTurn = not xTurn
            moves+=1
            #print('xT',xTurn)
```

```
			#At this point game is over, need to add to states list
			#a win is 1 a draw is 0 a loss is -1
			#if gameResult == -1:
				#print('loss')
			#print(gameStates)
			xTurns = len(gameStates)-1

			for x in range(0,len(gameStates)-1):
				#print(gameResult , decay , (x+1))
				gameLabels[xTurns-x] = gameResult * (decay ** (x))

			gameLabels[xTurns] = gameResult
			#print(gameLabels)
			for x in range(len(gameStates)):
				states.append(gameStates[x])
				labels.append(gameLabels[x])
		print('X,O,draw',xWins,oWins,draws)
		#for x in range(len(states)):
			#print(states[x],labels[x])
		return states, labels

def kerasStuff(states,labels):
		x_train = []
		for i in states:
			numB = lettersToNumerical(i)
			x_train.append(numB)
		x_train = np.asarray(x_train)
		x_train = x_train.astype('float32')
		y_train = np.asarray(labels)
		y_train = y_train.astype('float32')
		'''
		model = Sequential()
		model.add(Dense(9,activation = 'relu'))
		model.add(Dense(9,activation = 'relu'))
		model.add(Dense(5,activation = 'relu'))
		model.add(Dense(5,activation = 'relu'))
		model.add(BatchNormalization())
		model.add(Dense(3,activation = 'relu'))
		model.add(Dense(3,activation = 'relu'))
		model.add(Dense(1,activation = 'linear'))

		#sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
		#RMSprop
		model.compile(loss='mean_squared_error', optimizer='Adam')
		'''
		model = load_model('tictactoeFinal3.h5')
		model.fit(x_train, y_train, epochs=20, batch_size=16)
		model.save('tictactoeFinal4.h5')

def testAgent():
		#model = load_model('tictactoe2.h5')
		model = load_model('tictactoeTrain2.h5')
		#winningBoard = ['X','O',' ', 'X',' ',' ','X',' ','O']
		#wB = [np.asarray(lettersToNumerical(winningBoard)).astype('float32')]
		#wB = np.transpose(wB)
		#print(wB)
		#npwB = []
		#for x in range(len(wB)):
			#npwB.append(wB[x])
			#print(wB,npwB)
		#print(model.predict(wB))
		sP = np.array([[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]])
		print('sp',sP)
		startPredict= model.predict(sP)
```

```
        winPredict = model.predict(np.array([[0.4, -0.4, 0.0, 0.4, 0.0, 0.0, 0.4, 0.0, -0.4]]))
        losePredict = model.predict(np.array([[-0.4, 0.4, 0.0, -0.4, 0.0, 0.0, -0.4, 0.4, 0.4]]))
        print(winPredict)
        print(losePredict)
        print(startPredict)

def playAgent():
        model = load_model('tictactoeFinal4.h5')
        numGames = 200
        xWins = 0
        draws = 0
        oWins = 0


        allMoves = [0,1,2,3,4,5,6,7,8]
        for i in range(numGames):
                #print('Game ',i)

                board = createEmptyBoard()
                gameOver = False
                xTurn = True
                gameResult = 0
                moves = 0
                while gameOver!= True:

                        #print('Moves',moves)
                        if xTurn:
                                player = 'X'
                        else:
                                player = 'O'

                        if player == 'X':
                                #drawBoard(board)
                                bestValue = -99999
                                bestMove = None

                                for i in range(0, 9):
                                        if isSpaceFree(board,i):
                                                predictFormat = np.array([[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]])
                                                copy = createCopy(board)
                                                makeMove(copy,'X',i)
                                                numCopy = np.array(lettersToNumerical(copy))
                                                for j in range(0,9):
                                                        predictFormat[0][j] = numCopy[j]
                                                #print(predictFormat[0][1])
                                                #numCopy = [numCopy]
                                                #print(predictFormat)
                                                moveValue = model.predict(predictFormat)
                                                #print(moveValue)
                                                if moveValue > bestValue:
                                                        bestValue = moveValue
                                                        bestMove = i
                                move = bestMove
                        else:
                                #O will just be the AI
                                move = hardcodedAiMove(board,player)
                        #print('move',move)
                        #move = chooseRandomMoveFromList(board,allMoves)
                        if move == None:
                                #print('Game was a draw')
                                draws+=1
                                gameOver = True
                        else:
                                makeMove(board,player,move)
                                if isWinner(board,player):
                                        #print(player, ' won')
```

```
                    if player == 'X':
                        xWins+=1
                    else:
                        oWins+=1
                    gameOver = True
            xTurn = not xTurn
            moves+=1
    print('X:',xWins,'O:',oWins,'Draws:',draws)

def randomPlayer():

    numGames = 200
    xWins = 0
    draws = 0
    oWins = 0


    allMoves = [0,1,2,3,4,5,6,7,8]
    for i in range(numGames):
        #print('Game ',i)

        board = createEmptyBoard()
        gameOver = False
        xTurn = True
        gameResult = 0
        moves = 0
        while gameOver!= True:

            #print('Moves',moves)
            if xTurn:
                player = 'X'
            else:
                player = 'O'

            if player == 'X':
                move = chooseRandomMoveFromList(board,allMoves)
            else:
                #O will just be the AI
                move = hardcodedAiMove(board,player)
            #print('move',move)
            #move = chooseRandomMoveFromList(board,allMoves)
            if move == None:
                #print('Game was a draw')
                draws+=1
                gameOver = True
            else:
                makeMove(board,player,move)
                if isWinner(board,player):
                    #print(player, ' won')
                    if player == 'X':
                        xWins+=1
                    else:
                        oWins+=1
                    gameOver = True
            xTurn = not xTurn
            moves+=1
    print('X:',xWins,'O:',oWins,'Draws:',draws)

def hardcodedAiPlayer():
    numGames = 200
    xWins = 0
    draws = 0
    oWins = 0


    allMoves = [0,1,2,3,4,5,6,7,8]
```

```
	for i in range(numGames):
		#print('Game ',i)

		board = createEmptyBoard()
		gameOver = False
		xTurn = True
		gameResult = 0
		moves = 0
		while gameOver!= True:

			#print('Moves',moves)
			if xTurn:
				player = 'X'
			else:
				player = 'O'
			move = hardcodedAiMove(board,player)

			if move == None:
				#print('Game was a draw')
				draws+=1
				gameOver = True
			else:
				makeMove(board,player,move)
				if isWinner(board,player):
					#print(player, ' won')
					if player == 'X':
						xWins+=1
					else:
						oWins+=1
					gameOver = True
			xTurn = not xTurn
			moves+=1
	print('X:',xWins,'O:',oWins,'Draws:',draws)


board = createEmptyBoard()
drawBoard(board)
states = []
labels = []


#states, labels = createTrainingData()
#kerasStuff(states,labels)


#testAgent()
print('Agent:')
playAgent()
print('Random player')
randomPlayer()
```