Department of Computer Science
Technical University of Cluj-Napoca

Intelligent Systems

Laboratory activity 2018-2019

Multidimensional TicTacToe
easyAI

Name:Radu Beche
Group:30433
Email:beche.radu@gmail.com

Contents

Chapter 1

Tool

1.1 Tool description

EasyAI is an artificial intelligence framework for two-players abstract games such as Tic Tac Toe, Connect 4, Reversi, etc. It is written in Python and makes it easy to define the mechanisms of a game and play against the computer or solve the game.

1.2 Tool installation

The library is a python library that can be found as a pypl package and can be installed via the utility tool "pip". In order for you to install pip, you will have to install also Python, preferably Python 3.6.

Just run the command in a terminal session: "pip3 install easyai".

Also the tool can be installed by downloading the GitHub repository and running the command: "sudo python setup.py install" .

1.3 Tool - running examples

In the GitHub zoo repository, one can find sufficient examples to run directly. You only have to clone their repository to your local device. To name a few that I have run and had fun with are:

- Hexapawn
- Nim
- TicTacToe
- Cram
- Awele

To briefly describe an example: HexaPawn game was implemented in the TwoPlayerGames class provided by them. There was the board,the rules, the loss function and the scoring function. In the main function, Thre were instantiated the AI player and a HumanPlayer. The input was from the keyboard an it represented a move in form 'E6', like in chess.

The examples work pretty well but unfornately sometimes it is a little bit slow. The owner of the library encourages us to Cythonize as much code as we can in order to run faster.

1.4 Tool expressivity

The easyAI library offers different utility classes model that helped me a lot in my implementation, though, sometimes it was not enough so I had to extend in order to be compatible with other libraries as well.

- **TwoPlayerGames**: utility class that defines the game: where the players are stored, the score and where the "play" method if defined, method that allows us to play a game with 2 players. It is inherited by the class where we define our game with our own rules.
- **HumanPlayer**: it is the utility class that allows a human player to interact with the game via the keyboard. It also assures the correctness of the move that it is input.
- **AIPlayer**: it is a interface with the AI algorithms. It just calls the algorithms and ask for a move.
- **Negamax**: utility class where the Negamax algorithm is implemented by the library
- **SSS** : utility class where the SSS algorithm is implemented by the library
- **DUAL** : utility class where the Dual algorithm is implemented by the library

This are all the item that I have used from the library.

Chapter 2

Project

2.1 Narrative description

The aim of my project is to develop a multidimensional TicTacToe game that could be played against an AI. The dimensions of the games could be conigured by the user on 3 dimensions. The environment would provide the player with bots to play against. The user will be able to select the algorithm and the difficulty it plays against. Tic-tac-toe is a game for two players, X and O, who take turns marking the spaces in a m*n*k grid. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row wins the game.

2.2 Facts

The fats that can not be change are the rules of the game.

2.3 Specifications

The project will have the following specifications:

- provide the game environment, where all the rules are deffinied.
- provide multiple player option (computer vs computer, human vs human, human vs computer)
- provide 5 AI algorithms to play agains
- provide the option that the board size can be resized

2.4 Top level design of the scenario

The project scenario is as follow:

1. The player choose the board size
2. The player choose its opponent
3. If the opponent is a AI bot, it will also choose its difficulty
4. The game starts, and when the conditions, the env will show you the winner.

2.5 Knowledge acquisition

The algorithms that are in tool do not require any knowledge aquisition. The reinforcement learning agent requires data in order to be trained, but this data will be generatred with the other algorithms that the easyAi library provides.

The knowledge and informations that I require I will get from the various sites like:

- www.medium.com
- www.github.io
- www.openai.com

Chapter 3

Preliminary results

My preliminary result are that the implemented algotithms are very slow if you want a good adversary, and in order to be used for a mobile application the computation time should be reduced a lot. I will attempt to do so by implementin an Reinforcement learning algorithm.

I have noticed that the algorithms always provide the same move given a certain state(a state is a certain game state i.e. board). So in order to make it more real life like, I have changed the Negamax algorithm, such that, when multiple result have the same to choose randomly from them , rather then choosing the first one.

A better choice the language I have choosen i.e. Python would be C++, due to its compiled code. Also, the larger the size of the board, the longer it takes the algorithm to compute the next move, and for larger size, a normal computer is simply not enough.

Creating the game environment was not very hard, thus many online resources can be found online. In order to be able to generate and verify the moves for any board, I had to generalize the the generation process. This does not raised any difficulties.

The algorithms implemented and tested are:

- SSS
- DUAL
- Negamax
- Negamaxwith random choosing

Also, I have identified the need of using more specialized tools, so in the future, I will try to implement my solution using those tools such as:

- Numpy: will allow me to faster process array data and also allow me for a easier use of the ML applications
- Keras: will allow me to develop and train with minimum effort a reinforcement learning agent using a neural network as basis
- OpenAI Gym: is a framework that allow us to easily develop reinforcement learning agent

A way to visualize the result should be developed. I will try to find a tools suited for this task. One should be able to visualize the result in a plotted graph.

Chapter 4

Implementation details

In this section I am going to briefly describe the frameworks and libraries that I have used and how and exemplify with concludent examples.

- Numpy: allows me to faster process array data and also allow me for a easier use of the ML applications. It was of great use when when trying to store and process data for training. Being implement in C++ it is way faster than the standard python array. I also used it for computing certain mathematical values.
- Keras: I have used this library to define and train a neural network for a reinforcement learning agent
- OpenAI Gym: I used a class from this library in order to be able to better understand the RL agent
- Matplotlib : I have used this library to generate the plots used later in this documentation

4.1 Relevant code

4.2 Negamax

Below you there is the Negamax algorithm with alpha beta pruning implementation. Negamax search is a variant form of minimax search that relies on the zero-sum property of a two-player game.

This algorithm relies on the fact that max (a , b) = min (a , b) to simplify the implementation of the minimax algorithm. More precisely, the value of a position to player A in such a game is the negation of the value to player B. Thus, the player on move looks for a move that maximizes the negation of the value resulting from the move: this successor position must by definition have been valued by the opponent. The reasoning of the previous sentence works regardless of whether A or B is on move. This means that a single procedure can be used to value both positions. This is a coding simplification over minimax, which requires that A selects the move with the maximum-valued successor while B selects the move with the minimum-valued successor. Alpha/beta pruning and maximum search depth constraints in negamax can result in partial, inexact, and entirely skipped evaluation of nodes in a game tree.

```
def negamax(game, depth, origDepth, scoring, alpha=+inf, beta=-inf):
    if (depth == 0) or game.is_over():
        score = scoring(game)
        if score == 0:
```

```
            return score
        else:
            return  (score - 0.01*depth*abs(score)/score)

    possible_moves = game.possible_moves()

    state       =   game
    best_move   =   random.choice(possible_moves)

    if depth == origDepth:
        state.ai_move = best_move

    bestValue   = -inf
    unmake_move = hasattr(state, 'unmake_move')
    random.shuffle(possible_moves)

    for move in possible_moves:
        if not unmake_move:
            game = state.copy() # re-initialize move

        game.make_move(move)
        game.switch_player()

        move_alpha = - negamax(game, depth-1, origDepth, scoring,
                               -beta, -alpha, tt)

        if unmake_move:
            game.switch_player()
            game.unmake_move(move)

        if bestValue < move_alpha:
            bestValue = move_alpha
            best_move = move

        if  alpha < move_alpha :
            alpha = move_alpha
            if depth == origDepth:
                state.ai_move = move
            if (alpha >= beta):
                break

    return bestValue
```

4.3 RL Agent

Q-learning is a reinforcement learning technique used in machine learning. The goal of Q-learning is to learn a policy, which tells an agent what action to take under what circumstances.

The DQN Reinforcement learning is unstable or divergent when a nonlinear function approximator such as a neural network is used to represent Q. This instability comes from the

correlations present in the sequence of observations, the fact that small updates to Q may significantly change the policy and the data distribution, and the correlations between Q and the target values.

Reinforcement learning involves an agent, a set of states S, and a set A of actions per state. By performing an action , the agent transitions from state to state. Executing an action in a specific state provides the agent with a reward (a numerical score).

The weight for a step from a state t steps into the future is calculated as Y - t. Y (the discount factor) is a number between 0 and 1 and has the effect of valuing rewards received earlier higher than those received later (reflecting the value of a "good start"). Y may also be interpreted as the probability to succeed (or survive) at every step t.

The algorithm, therefore, has a function that calculates the quality of a state-action combination.

Before learning begins, Q is initialized to a possibly arbitrary fixed value (chosen by the programmer). Then, at each time t the agent selects an action a t, observes a reward r , enters a new state s t + 1 s(t+1) (that may depend on both the previous state s(t) and the selected action), and Q is updated. The core of the algorithm is a simple value iteration update, using the weighted average of the old value and the new information.

The goal of the agent is to maximize its total (future) reward. It does this by adding the maximum reward attainable from future states to the reward for achieving its current state, effectively influencing the current action by the potential future reward. This potential reward is a weighted sum of the expected values of the rewards of all future steps starting from the current state.

Chapter 5

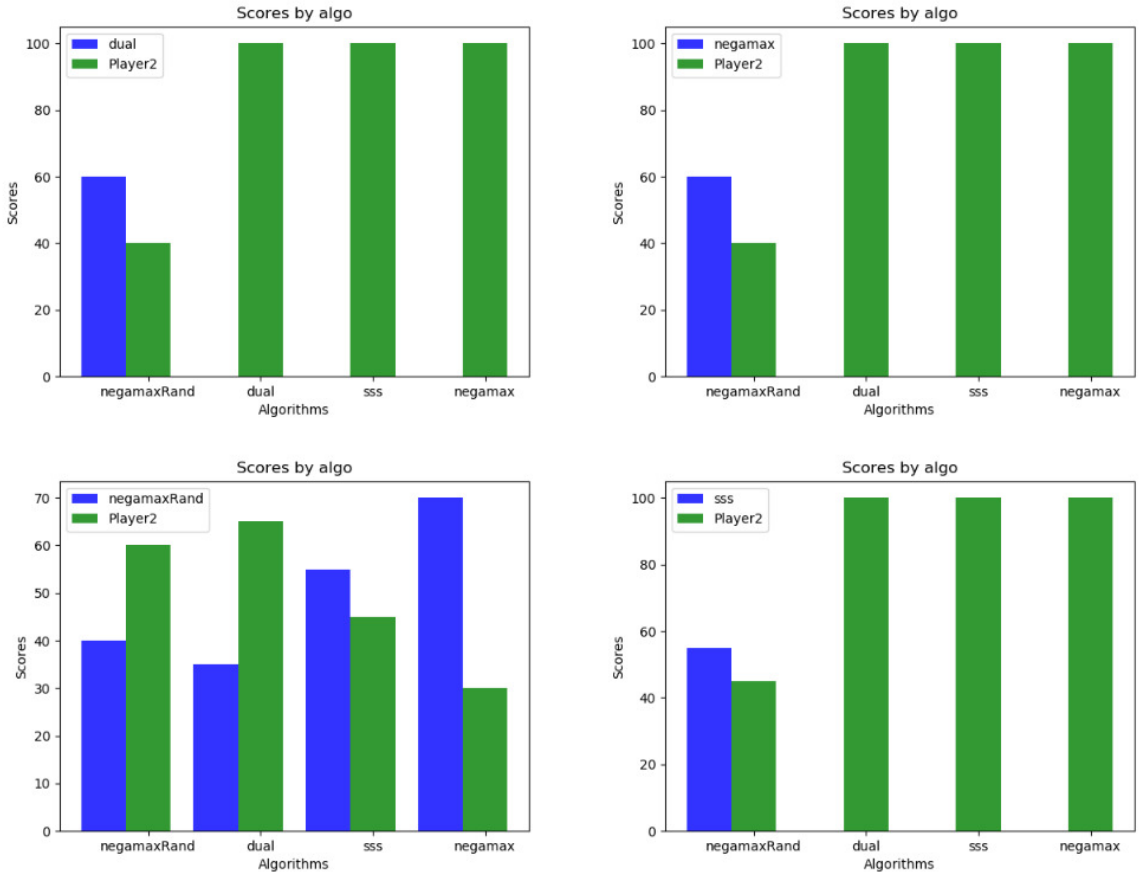
Graphs and experiments

5.1 Evaluation metrics

First let me introduce you to my work. I will compare 5 algorithms: Negamax with alpha-beta prunning, a modified version of the precedent specified algorithm , DUAL, SSS and a reinforcement learning agent trained with all the above algorithms with various difficulty rating. By difficulty I understand the number of steps that the standard AI algo thinks in advance.

The way that I have chosen to evaluate my work is pretty straight forward: put each algorithm to fight with all the others. The result were noted after 20 rounds, though for a better precision I could have opted for a larger number of sample, but the computing power of my laptop is pretty limited.

The result are as follow:



The algorithms are as following:

- top-left: DUAL
- top-right: Negamax
- bottom-left: Negamax Random
- bottom-right: SSS

One can conclude that the aditonal modification that I have made improved the overall performance and unpredictibility of the algorithm eventhough it is as first or the second player.

I could not test the reinforcement agent thus my laptop does not have the required computational power so I will show you a demo at the lab.

But, from my observation, I can conclude that the DQN is not properly suited for this kind of application. Further details can be found in the below section, i.e Related work and documentation.

Chapter 6

Related work and documentation

6.1 Related approaches

On very popular approach that has released not very long time ago is and it is called AlphaGo. AlphaGo uses a Monte Carlo tree search algorithm to find its moves based on knowledge previously "learned" by machine learning, specifically by an artificial neural network (a deep learning method) by extensive training, both from human and computer play. A neural network is trained to predict AlphaGo's own move selections and also the winner's games. This neural net improves the strength of tree search, resulting in higher quality of move selection and stronger self-play in the next iteration.

6.2 Advantages and limitations of your solution

The classical AI solution provide the following:

- advantages:
  - very good for small games, with few possible moves
  - relatively simple to code and use
  - does not need any information for training
  - very memory efficient
  - can slightly change the board size
- disadvantages:
  - very slow for large possible sets of moves
  - very computational expensive
  - can not improve over time

The RL agent solution provide the following:

- advantages:
  - can get very very good
  - computational inexpensive
  - faster for larger board sizes

  - can train directly from played games
- disadvantages:
  - memory inefficient for larger models
  - can not modify the action space
  - needs a lot of training
  - needs lots of data for training

Chapter 7

Possible extensions of the current work

7.1 Algorithm

One possible extension of my work would be to implement the AlphaGo aproach. I think that that is the best algorithm for an m*n*k game. And also I would try to make it a little bit more efficient, in order to work for mobile phones.

7.2 Bussiness oriented

I would try to make my solution more appealing in order to present it to a client. I would try to accentuate the fact that I would be able to make a bot for evey game they want. I think that making games equipped with very powerfull opponent would be a great advantage over the other games.

Chapter 8

Results dissemination

8.1 Self-assessment

In my opinion I would say the I have mastered my tool and succesfully developed a game and a AI bot for him. Though, I am not satisfied with the time it takes the algorithm to compute the next move, and also I would try to make a more general solution for my Reinforcement learning agent. I have learned a lot, and I will deffinetly continuing studying this domain thus I think this is the future.

Appendix A

Your original code

This section should contain only code developed by you, without any line re-used from other sources. This section helps me to correctly evaluate your amount of work and results obtained. Including in this section any line of code taken from someone else leads to failure of IS class this year. Failing or forgetting to add your code in this appendix leads to grade 1. Don't remove the above lines.

Listing A.1: Insert code directly in your document

RLAgent.py

```
import keras
import numpy as np

from keras import models

#====================================== RL agent ==================
class RLAgent:
    '''This class aims to work be an interface for a keras model with the easyAI library.

    '''
    #====================================== INIT ==================
    def __init__(self, modelPath):
        '''Constructor

        Arguments:
            modelPath {str} -- the path of the model(.h5 file)
        '''

        self.model = models.load_model(modelPath)

    #====================================== PREDICT ==================
    def predict(self, state):
        '''Make a prediction with the current model

        Arguments:
            state {[type]} -- the type has to exact like that wich you have trained your
            model

        Returns:
            [type] -- the prediction of the model
        '''

        return self.model.predict(state)
```

```
    #========================================== __CALL__ ============
    def __call__(self, game, normVal = 2):
        '''Used in easyAi library

        Arguments:
            game {TwoPlayerGame} -- the game

        Keyword Arguments:
            normVal {int} -- the normalization value for the board(if necessary) (default:

        Returns:
             int -- the best possible move
        '''

        predictBoard = np.array(game.board) / 2

        pred    =    self.predict([[predictBoard]])

        # get the best possible move
        while True:
            predPos = np.argmax(pred) + 1
            if predPos in game.possible_moves():
                break
            pred[0][predPos-1] = -1

        return predPos
```

MoveGenerator.py

```
import numpy as np

#========================================== GENERATE CUBE ============
def generateCube(xLines, yLines, zLines):
    cube = np.zeros((xLines, yLines, zLines), dtype='int')
    index = 1
    for xAxis in range(xLines):
        for yAxis in range(yLines):
            for zAxis in range(zLines):
                cube[xAxis][yAxis][zAxis] = index
                index += 1
    return cube

#========================================== GENERATE LIST ============
def generateLoseList(boardSize, winningInRow):
    '''
    Generate the lose move for a 3d TicTacToe game

    Arguments:
        boardSize {(int,int,int)} -- the size on x,y and z axis
        winningInRow {int} -- the number of marks required to win
    '''
    assert boardSize[0] >= 1, 'xLines can not be less than 3'
    assert boardSize[1] >= 1, 'yLines can not be less than 3'
    assert boardSize[2] >= 1, 'zLines can not be less than 1'
    xLines = boardSize[0]
    yLines = boardSize[1]
    zLines = boardSize[2]

    cube = generateCube(xLines, yLines, zLines)
    moves = []
```

```
#z axis
for j in range(yLines):
    for k in range(zLines):
        for i in range(xLines - winningInRow +1):
            aux = []
            for o in range(winningInRow):
                aux.append(cube[i+o][j][k])
            moves.append(aux)
#x axis
for i in range(xLines):
    for j in range(yLines):
        for k in range(zLines - winningInRow +1):
            moves.append(cube[i][j][k:k+winningInRow].tolist())

#y axis
for i in range(xLines):
    for k in range(zLines):
        for j in range(yLines - winningInRow +1):
            aux = []
            for o in range(winningInRow):
                aux.append(cube[i][j+o][k])
            moves.append(aux)



#xy axis right
for i in range(xLines):
    for j in range(yLines- winningInRow +1):
        for k in range(zLines- winningInRow +1):
                aux = []
                for o in range(winningInRow):
                    aux.append(cube[i][j+o][k+o])
                moves.append(aux)

#xy axisleft
for i in range(xLines):
    for j in range(yLines-winningInRow+1):
        for k in range(winningInRow-1, zLines):
            aux = []
            for o in range(winningInRow):
                aux.append(cube[i][j+o][k-o])
            moves.append(aux)

#xz axis right
for j in range(yLines):
    for k in range(zLines- winningInRow +1):
        for i in range(xLines- winningInRow +1):
                aux = []
                for o in range(winningInRow):
                    aux.append(cube[i+o][j][k+o])
                moves.append(aux)

#xz axis left
for j in range(yLines):
    for i in range(xLines - winningInRow +1 ):
        for k in range(winningInRow -1, zLines):
                aux = []
                for o in range(winningInRow):
                    aux.append(cube[i+o][j][k-o])
```

```
                    moves.append(aux)

        #yz axis right
        for k in range(zLines):
            for j in range(yLines- winningInRow +1):
                for i in range(xLines- winningInRow +1):
                        aux = []
                        for o in range(winningInRow):
                            aux.append(cube[i+o][j+o][k])
                        moves.append(aux)

        #yz axis rightfor k in range(zLines):
            for i in range(xLines- winningInRow +1):
                for j in range(winningInRow - 1, yLines):
                        aux = []
                        for o in range(winningInRow):
                            aux.append(cube[i+o][j-o][k])
                        moves.append(aux)

        assert len(moves) != 0, 'There are no possible winning moves. please check winningInRow
        return moves
```

TicTacToe3D.py

```
import random
import numpy                    as  np
import game.MoveGenerator       as  mg

from game.TwoPlayerGame         import  TwoPlayersGame

from gym import Env

class TicTacToe3D( TwoPlayersGame, Env ):
    '''A class that implements the rules of a tick tack toe game. It enherits the easyAI
    class (due to my project for university) and the Env class from openAI gym.

    Arguments:
        TwoPlayersGame {[type]} -- [description]
        Env {[type]} -- [description]
    '''


#========================================= CONSTRUCTOR ==========
    def __init__(self,winningInRow, boardSize, players, scoringType):
        '''Constructor of the TiTacToe3D game. Here the basic rules are deffinied

        Arguments:
            winningInRow {int} -- the number of conssecutive boxes required to win
            boardSize {[int,int,int]} -- a tuple with 3 elements for board size
            players {Player} -- the player
            scoringType {int} -- the coring type
        '''
        self.verbose = False
        self.players    =   players
        self.nplayer    =   1

        self.lose_move  =   mg.generateLoseList(boardSize,winningInRow)

        self.winningInRow = winningInRow
```

```
        self.boardSize  =   boardSize
        self.boardLen   =   boardSize[0] * boardSize[1] * boardSize[2]
        self.board      =   [0 for i in range(self.boardLen)]
        self.last_move  =   0

        self.scoringType = scoringType
        self.copy = self

#================================== RESET ==================================
    def reset(self):
        '''REINITIALIZE THE CURRENT PLAYER AND THE BOARD TO the initial status

        Returns:
            [tuple] -- return the board
        '''

        self.nplayer    =   1
        self.board      =   [0 for i in range(self.boardLen)]
        self.last_move  =   0

        return self.board

#================================== STEP ==================================
    def step(self, action):
        '''step
        it make a move, inherited from the openai gym environment

        Arguments:
            action {int} -- the move number to be done

        Returns:
            board {tuple} -- the current board, after the action was done
            score {tuple} -- the score/reward that was obtain after the move
            is_over{bool} -- check to see if the games is finished
        '''

        self.make_move(action)
        self.switch_player()
        score = self.scoring()

        return self.board, score, self.is_over(), {}

#================================== POSSIBLE MOVES ==========
    def possible_moves(self):
        '''All the possible moves that can be done on the board

        Returns:
            List -- the list with all possible moves
        '''

        lst = [i+1 for i,e in enumerate(self.board) if e==0]
        return lst

#================================== MAKE MOVE ==============
    def make_move(self, move):
        '''Make move

        Arguments:
            move {int} -- the move that will be done
        '''
```

```
        self.board[int(move)-1] = self.nplayer

    #============================== UNMAKE MOVE ==========
    def unmake_move(self, move):
        '''Make a box to 0. It speeds up the algorithms

        Arguments:
            move {int} -- the move that will be unmake
        '''

        self.board[int(move)-1] = 0

    #============================== LOSE ==================
    def lose(self):
        '''Check is oponent has lost

        Returns:
            bool -- -
        '''

        for move in self.lose_move:
            if self.board[move[0]-1] == self.board[move[1]-1] == self.board[move[2]-1] ==
                return True
        return False

    #============================== WIN ===================
    def win(self):
        '''Check if the opponent has won

        Returns:
            bool -- -
        '''
        for move in self.lose_move:
            if self.board[move[0]-1] == self.board[move[1]-1] == self.board[move[2]-1] ==
                return True
        return False

    #============================== IS_OVER ===============
    def is_over(self):
        '''Check to see if anybody has won or if there are no more moves to make.

        Returns:
                bool -- if the game is over or not
        '''

        return (self.possible_moves() == []) or self.lose()

    #============================== SHOW ==================
    def show(self):
        '''
            Display the current board
        '''

        index = 0
        for i in range(self.boardSize[0]):
            for j in range(self.boardSize[1]):
                for k in range(self.boardSize[2]):
                    if self.board[index] == 0:
                        print('.', end = ' ')
```

```
                elif self.board[index] == 1:
                    print('X', end = ' ')
                else:
                    print('O', end = ' ')
                index += 1
            print()
        print('')

#=================================== SCORING ================
    def scoring(self):
        '''The scoring type applied for the AI algorythm

        Raises:
            AttributeError -- when the scoring type is not correctly deffinied

        Returns:
            int -- score
        '''

        #offensive
        if self.scoringType == 0:
            if self.lose():
                return -100
            elif self.win():
                return 1000

            return 10

        #deffensive
        elif self.scoringType == 1:
            if self.lose():
                return -1000
            elif self.win():
                return 100
            return 10

        #normal
        elif self.scoringType == 2:
            if self.lose():
                return -105
            elif self.win():
                return 100
            return 10

        #not win
        elif self.scoringType == 3:
            if self.lose():
                return -1000
            return 0

        #normal reinf
        elif self.scoringType == 4:
            if self.lose():
                return -1
            elif self.win():
                return 1
            return -0.05

        else:
            raise AttributeError("Scoring not corectly configured")
```

```
# Script used for training the RL agent. It randoply trains the agent as both, 1st player,
# and second player. Also, it randomly introduces random moves. It uses multiple algorithm
# to train against. It saves the model every
# 500 episodes.
#
# Note: this RL agent is not as good as you think. A way better combination would bve an
# AlphaGo like approach

import gym
import copy
import keras
import random
import numpy as np

from keras.models import Sequential
from keras.layers import InputLayer, Dense, Dropout

from game.TicTacToe3D    import TicTacToe3D
from game.Player         import AI_Player, Human_Player
from easyAI              import DUAL, SSS, Negamax
from aiAlgos.Negamax     import NegamaxRand

#================================== MODEL DEFINITION ==================
model = Sequential()
model.add( Dense(10,      activation = 'relu', input_dim = 9))
model.add( Dense(100,     activation = 'relu') )
model.add( Dense(100,     activation = 'relu') )
model.add( Dense(9,       activation = 'linear') )

model.compile(loss='mse', optimizer='adam', metrics=['mae'])

# model = keras.models.load_model('./models/mix/train_49500.h5')
#================================== PARAMETERS ========================

env             =   TicTacToe3D(3, (1,3,3), [Human_Player(),Human_Player() ], 4)
num_episodes    =   100000
y               =   0.95
rlFirst         =   True
decay           =   0.9999
aiAlgos         =   [SSS(5), NegamaxRand(4), NegamaxRand(2), Negamax(2)]

#================================== TRAINING LOOP =====================
print('Start training')
for i in range(num_episodes):

    #reset the environment
    state       =   env.reset()
    done        =   False

    #get the training conditions
    rlFirst         =   random.choice([True,False])
    trainAlgo       =   random.choice(aiAlgos)
    trainPlayer     =   AI_Player(trainAlgo)

    y *= decay                                          #it learn slower after many iterat

    #================================== GAME LOOP =====================
    #play a game until it is finished
    while not done:
```

```
    # opponent move
    if not rlFirst:
        if random.random() > 0.1:
            trainMove           =   trainPlayer.ask_move(env)
        else:
            trainMove = random.choice(env.possible_moves())

        newState,_,done,_   =   env.step(trainMove)
        state               =   newState

        #debug message once every 500 episodes
        if done:
            if i % 500 == 0:
                print('EasyAI ended')
            break


    normalizedState =   np.array(copy.deepcopy(state))/2
    pred            =   model.predict([[normalizedState]])

    # get the best possible move
    for imoves in range(9):
        predPos = np.argmax(pred) + 1
        if predPos in env.possible_moves():
            break
        pred[0][predPos-1] = -0.1

    # make the step with the predicted move
    newState, reward, done, _   =   env.step(predPos)
    normalizedNewState          =   np.array(copy.deepcopy(newState))/2

    # compute label for training
    target                      =   reward + y * np.max(model.predict([[normalizedNewS
    targetVec                   =   model.predict([[normalizedState]])
    targetVec[0][predPos-1]     =   target

    # train the model
    model.fit([[normalizedState]], targetVec, epochs=1, verbose=0)

    #debug message
    if done:
        if i % 500 == 0:
            print('RL has ended')
        break

    state =   newState

    # opponent move
    if rlFirst:
        if random.random() > 0.1:
            trainMove   =     trainPlayer.ask_move(env)
        else:
            trainMove   =     random.choice(env.possible_moves())

        newState,_,done,_   =   env.step(trainMove)

    state =   newState

#debug message
```

```
    if i % 500 == 0:
        print("Episode {} of {} , rlFirst {}".format(i, num_episodes, rlFirst))
        print(state)

        #save the model
        keras.models.save_model(model, './mix3/train_'+str(i)+'.h5')

from aiAlgos.RLAgent      import      RLAgent
from game.Player          import      Human_Player, AI_Player
from game.TicTacToe3D     import      TicTacToe3D
from aiAlgos.Negamax      import      NegamaxRand
from easyAI               import      DUAL, SSS, Negamax

# classical algos /easyAi
steps       =   4                                                   #the number of steps AI th
algos       =   [   DUAL(steps),                                    #0
                    Negamax(steps),                                 #1
                    NegamaxRand(steps),                             #2 - when states are equal
                    SSS(steps),                                     #3
                    RLAgent('./models/mix2/train_17500.h5')         #4 - trained on a (1,3,3)
                    ]

#select choice from above
choice      =   0
aiAlgo      =   algos[choice]

#========================================= PLAYERS ================
pl1     =   AI_Player(aiAlgo)
pl2     =   Human_Player()

#========================================= GAME ================
env     =   TicTacToe3D(3, (1,3,3), [pl1, pl2], 4)
env.play()

import pickle
import random
inf = float('infinity')

DISCLAIMER! ... THIS IS JUST A MODIFIED PART

#========================================= NEGAMAX RAND ==========
def negamax(game, depth, origDepth, scoring, alpha=+inf, beta=-inf, tt=None):
    if (depth == 0) or game.is_over():
        score = scoring(game)
        if score == 0:
            return score
        else:
            return  (score - 0.01*depth*abs(score)/score)

    possible_moves = game.possible_moves()

    state       =   game
    best_move   =   random.choice(possible_moves)

    if depth == origDepth:
        state.ai_move = best_move

    bestValue   = -inf
    unmake_move = hasattr(state, 'unmake_move')
    random.shuffle(possible_moves)
```

```
    for move in possible_moves:
        if not unmake_move:
            game = state.copy() # re-initialize move

        game.make_move(move)
        game.switch_player()

        move_alpha = - negamax(game, depth-1, origDepth, scoring,
                               -beta, -alpha, tt)

        if unmake_move:
            game.switch_player()
            game.unmake_move(move)

        if bestValue < move_alpha:
            bestValue = move_alpha
            best_move = move

        if  alpha < move_alpha :
            alpha = move_alpha
            if depth == origDepth:
                state.ai_move = move
            if (alpha >= beta):
                break

    return bestValue

    from copy import deepcopy

class TwoPlayersGame:

    def play(self, nmoves=1000, verbose=True):
        history = []

        if verbose:
            self.show()

        for self.nmove in range(1, nmoves + 1):

            if self.is_over():
                if(len(self.possible_moves()) == 0):
                    print("Draw! :(")
                    break

                if verbose:
                    print("Player %d won!" % (self.nopponent))
                break

            move = self.player.ask_move(self)
            history.append((deepcopy(self), move))
            self.make_move(move)

            if verbose:
                print("\nMove #%d: player %d plays %s :" % (
                      self.nmove, self.nplayer, str(move)))
                self.show()

            self.switch_player()
```

```
        history.append(deepcopy(self))
        return history
```

Appendix B

Quick technical guide for running your project

Requirements

Step by step technical manual

If you just want to:

1. Play the TicTacToe:
   (a) run PlayTicTacToe.py
   (b) you can open and configure different parameters such as: how many steps in advance the AI thinks, against which algorithm will you play, and also chose the model for the reinforcement learning agent.
2. Train the agent:
   (a) run ReinforcementAgentTrainer.py
   (b) with this script you can define a new RL agent or to train an existing one. I encourage to take a look, thus it is very intuitive.

Appendix C

Check list

1. Your original code is included in the Appendix .
2. Your original code and figures are readable.
3. All the references are added in the Bibliography section.
4. All your figures are referred in text (with command ref), described in the text, and they have relevant caption.
5. The final documentation describes only your project. Don't forget to remove all tutorial lines in the template (like these one).
6. The main algorithm of your tool is formalised in latex in chapter **??**.

Bibliography