In [1]:
```
import math
import numpy as np
import time
from __future__ import print_function
import sys
import tensorflow as tf
from collections import deque
import shutil
import random
import os
sys.setrecursionlimit(5000)
class dotdict(dict):
    def __getattr__(self, name):
        return self[name]

args = {
    'numIters': 1000,
    'numEps': 100,
    'tempThreshold': 15,
    'updateThreshold': 0.6,
    'maxlenOfQueue': 200000,
    'numMCTSSims': 25,
    'arenaCompare': 40,
    'cpuct': 1,

    'checkpoint': './temp/',
    'load_model': False,
    'load_folder_file': ('/models','best.pth.tar'),
    'lr': 0.001,
    'dropout': 0.3,
    'epochs': 10,
    'batch_size': 64,
    'num_channels': 512,
}

class MCTS():
    """
    This class handles the MCTS tree.
    """

    def __init__(self, game, nnet, args):
        self.game = game
        self.nnet = nnet
        self.args = args
        self.Qsa = {}       # stores Q values for s,a (as defined in the pap
        self.Nsa = {}       # stores #times edge s,a was visited
        self.Ns = {}        # stores #times board s was visited
        self.Ps = {}        # stores initial policy (returned by neural net)

        self.Es = {}        # stores game.getGameEnded ended for board s
        self.Vs = {}        # stores game.getValidMoves for board s

    def getActionProb(self, canonicalBoard, temp=1):
        """
        This function performs numMCTSSims simulations of MCTS starting from
        canonicalBoard.
```

```python
        Returns:
            probs: a policy vector where the probability of the ith action i
                   proportional to Nsa[(s,a)]**(1./temp)
        """
        for i in range(self.args['numMCTSSims']):
            self.search(canonicalBoard)

        s = self.game.stringRepresentation(canonicalBoard)
        counts = [self.Nsa[(s,a)] if (s,a) in self.Nsa else 0 for a in range

        if temp==0:
            bestA = np.argmax(counts)
            probs = [0]*len(counts)
            probs[bestA]=1
            return probs

        counts = [x**(1./temp) for x in counts]
        probs = [x/float(sum(counts)) for x in counts]
        return probs


    def search(self, canonicalBoard):
        """
        This function performs one iteration of MCTS. It is recursively call
        till a leaf node is found. The action chosen at each node is one tha
        has the maximum upper confidence bound as in the paper.

        Once a leaf node is found, the neural network is called to return an
        initial policy P and a value v for the state. This value is propogat
        up the search path. In case the leaf node is a terminal state, the
        outcome is propogated up the search path. The values of Ns, Nsa, Qsa
        updated.

        NOTE: the return values are the negative of the value of the current
        state. This is done since v is in [-1,1] and if v is the value of a
        state for the current player, then its value is -v for the other pla

        Returns:
            v: the negative of the value of the current canonicalBoard
        """

        s = self.game.stringRepresentation(canonicalBoard)

        if s not in self.Es:
            self.Es[s] = self.game.getGameEnded(canonicalBoard, 1)
        if self.Es[s]!=0:
            # terminal node
            return -self.Es[s]

        if s not in self.Ps:
            # leaf node
            self.Ps[s], v = self.nnet.predict(canonicalBoard)
            valids = self.game.getValidMoves(canonicalBoard, 1)
            self.Ps[s] = self.Ps[s]*valids      # masking invalid moves
            self.Ps[s] /= np.sum(self.Ps[s])    # renormalize
```

```
            self.Vs[s] = valids
            self.Ns[s] = 0
            return -v

        valids = self.Vs[s]
        cur_best = -float('inf')
        best_act = -1

        # pick the action with the highest upper confidence bound
        for a in range(self.game.getActionSize()):
            if valids[a]:
                if (s,a) in self.Qsa:
                    u = self.Qsa[(s,a)] + self.args['cpuct']*self.Ps[s][a]*m
                else:
                    u = self.args['cpuct']*self.Ps[s][a]*math.sqrt(self.Ns[s

                if u > cur_best:
                    cur_best = u
                    best_act = a

        a = best_act
        next_s, next_player = self.game.getNextState(canonicalBoard, 1, a)
        next_s = self.game.getCanonicalForm(next_s, next_player)

        v = self.search(next_s)

        if (s,a) in self.Qsa:
            self.Qsa[(s,a)] = (self.Nsa[(s,a)]*self.Qsa[(s,a)] + v)/(self.Ns
            self.Nsa[(s,a)] += 1

        else:
            self.Qsa[(s,a)] = v
            self.Nsa[(s,a)] = 1

        self.Ns[s] += 1
        return -v


class Arena():
    """
    An Arena class where any 2 agents can be pit against each other.
    """
    def __init__(self, player1, player2, game, display=None):
        """
        Input:
            player 1,2: two functions that takes board as input, return acti
            game: Game object
            display: a function that takes board as input and prints it (e.g
                     display in othello/OthelloGame). Is necessary for verbo
                     mode.

        see othello/OthelloPlayers.py for an example. See pit.py for pitting
        human players/other baselines with each other.
        """
        self.player1 = player1
        self.player2 = player2
        self.game = game
```

```
        self.display = display

    def playGame(self, verbose=False):
        """
        Executes one episode of a game.

        Returns:
            either
                winner: player who won the game (1 if player1, -1 if player2
            or
                draw result returned from the game that is neither 1, -1, no
        """
        players = [self.player2, None, self.player1]
        curPlayer = 1
        board = self.game.getInitBoard()
        it = 0
        while self.game.getGameEnded(board, curPlayer)==0:
            it+=1
            if verbose:
                assert(self.display)
                print("Turn ", str(it), "Player ", str(curPlayer))
                self.display(board)
            action = players[curPlayer+1](self.game.getCanonicalForm(board, 

            valids = self.game.getValidMoves(self.game.getCanonicalForm(boar

            if valids[action]==0:
                print(action)
                assert valids[action] >0
            board, curPlayer = self.game.getNextState(board, curPlayer, acti
        if verbose:
            assert(self.display)
            print("Game over: Turn ", str(it), "Result ", str(self.game.getG
            self.display(board)
        return self.game.getGameEnded(board, 1)

    def playGames(self, num, verbose=False):
        """
        Plays num games in which player1 starts num/2 games and player2 star
        num/2 games.

        Returns:
            oneWon: games won by player1
            twoWon: games won by player2
            draws:  games won by nobody
        """
#         end = time.time()
        print('in playGames')
        eps = 0
        maxeps = int(num)

        num = int(num/2)
        oneWon = 0
        twoWon = 0
        draws = 0
        for _ in range(num):
            gameResult = self.playGame(verbose=verbose)
```

```
            if gameResult==1:
                oneWon+=1
            elif gameResult==-1:
                twoWon+=1
            else:
                draws+=1
            # bookkeeping + plot progress
            eps += 1
            #eps_time.update(time.time() - end)
#             end = time.time()

        self.player1, self.player2 = self.player2, self.player1

        for _ in range(num):
            gameResult = self.playGame(verbose=verbose)
            if gameResult==-1:
                oneWon+=1
            elif gameResult==1:
                twoWon+=1
            else:
                draws+=1
            # bookkeeping + plot progress
            eps += 1
#             eps_time.update(time.time() - end)
            end = time.time()

        return oneWon, twoWon, draws
class Board():

    # list of all 8 directions on the board, as (x,y) offsets
    __directions = [(1,1),(1,0),(1,-1),(0,-1),(-1,-1),(-1,0),(-1,1),(0,1)]

    def __init__(self, n):
        "Set up initial board configuration."

        self.n = n
        # Create the empty board array.
        self.pieces = [None]*self.n
        for i in range(self.n):
            self.pieces[i] = [0]*self.n

        # Set up the initial 4 pieces.
        self.pieces[int(self.n/2)-1][int(self.n/2)] = 1
        self.pieces[int(self.n/2)][int(self.n/2)-1] = 1
        self.pieces[int(self.n/2)-1][int(self.n/2)-1] = -1;
        self.pieces[int(self.n/2)][int(self.n/2)] = -1;

    # add [][] indexer syntax to the Board
    def __getitem__(self, index):
        return self.pieces[index]

    def countDiff(self, color):
        """Counts the # pieces of the given color
        (1 for white, -1 for black, 0 for empty spaces)"""
        count = 0
        for y in range(self.n):
            for x in range(self.n):
```

```
                if self[x][y]==color:
                    count += 1
                if self[x][y]==-color:
                    count -= 1
        return count

    def get_legal_moves(self, color):
        """Returns all the legal moves for the given color.
        (1 for white, -1 for black
        """
        moves = set()  # stores the legal moves.

        # Get all the squares with pieces of the given color.
        for y in range(self.n):
            for x in range(self.n):
                if self[x][y]==color:
                    newmoves = self.get_moves_for_square((x,y))
                    moves.update(newmoves)
        return list(moves)

    def has_legal_moves(self, color):
        for y in range(self.n):
            for x in range(self.n):
                if self[x][y]==color:
                    newmoves = self.get_moves_for_square((x,y))
                    if len(newmoves)>0:
                        return True
        return False

    def get_moves_for_square(self, square):
        """Returns all the legal moves that use the given square as a base.
        That is, if the given square is (3,4) and it contains a black piece,
        and (3,5) and (3,6) contain white pieces, and (3,7) is empty, one
        of the returned moves is (3,7) because everything from there to (3,4
        is flipped.
        """
        (x,y) = square

        # determine the color of the piece.
        color = self[x][y]

        # skip empty source squares.
        if color==0:
            return None

        # search all possible directions.
        moves = []
        for direction in self.__directions:
            move = self._discover_move(square, direction)
            if move:
                # print(square,move,direction)
                moves.append(move)

        # return the generated move list
        return moves

    def execute_move(self, move, color):
```

```
        """Perform the given move on the board; flips pieces as necessary.
        color gives the color pf the piece to play (1=white,-1=black)
        """

        #Much like move generation, start at the new piece's square and
        #follow it on all 8 directions to look for a piece allowing flipping

        # Add the piece to the empty square.
        # print(move)
        flips = [flip for direction in self.__directions
                      for flip in self._get_flips(move, direction, color)]
        assert len(list(flips))>0
        for x, y in flips:
            #print(self[x][y],color)
            self[x][y] = color

    def _discover_move(self, origin, direction):
        """ Returns the endpoint for a legal move, starting at the given ori
        moving by the given increment."""
        x, y = origin
        color = self[x][y]
        flips = []

        for x, y in Board._increment_move(origin, direction, self.n):
            if self[x][y] == 0:
                if flips:
                    # print("Found", x,y)
                    return (x, y)
                else:
                    return None
            elif self[x][y] == color:
                return None
            elif self[x][y] == -color:
                # print("Flip",x,y)
                flips.append((x, y))

    def _get_flips(self, origin, direction, color):
        """ Gets the list of flips for a vertex and direction to use with th
        execute_move function """
        #initialize variables
        flips = [origin]

        for x, y in Board._increment_move(origin, direction, self.n):
            #print(x,y)
            if self[x][y] == 0:
                return []
            if self[x][y] == -color:
                flips.append((x, y))
            elif self[x][y] == color and len(flips) > 0:
                #print(flips)
                return flips

        return []

    @staticmethod
    def _increment_move(move, direction, n):
        # print(move)
```

```python
        """ Generator expression for incrementing moves """
        move = list(map(sum, zip(move, direction)))
        #move = (move[0]+direction[0], move[1]+direction[1])
        while all(map(lambda x: 0 <= x < n, move)):
        #while 0<=move[0] and move[0]<n and 0<=move[1] and move[1]<n:
            yield move
            move=list(map(sum,zip(move,direction)))
            #move = (move[0]+direction[0],move[1]+direction[1])

class OthelloGame():
    def __init__(self, n):
        self.n = n

    def getInitBoard(self):
        # return initial board (numpy board)
        b = Board(self.n)
        return np.array(b.pieces)

    def getBoardSize(self):
        # (a,b) tuple
        return (self.n, self.n)

    def getActionSize(self):
        # return number of actions
        return self.n*self.n + 1

    def getNextState(self, board, player, action):
        # if player takes action on board, return next (board,player)
        # action must be a valid move
        if action == self.n*self.n:
            return (board, -player)
        b = Board(self.n)
        b.pieces = np.copy(board)
        move = (int(action/self.n), action%self.n)
        b.execute_move(move, player)
        return (b.pieces, -player)

    def getValidMoves(self, board, player):
        # return a fixed size binary vector
        valids = [0]*self.getActionSize()
        b = Board(self.n)
        b.pieces = np.copy(board)
        legalMoves =  b.get_legal_moves(player)
        if len(legalMoves)==0:
            valids[-1]=1
            return np.array(valids)
        for x, y in legalMoves:
            valids[self.n*x+y]=1
        return np.array(valids)

    def getGameEnded(self, board, player):
        # return 0 if not ended, 1 if player 1 won, -1 if player 1 lost
        # player = 1
        b = Board(self.n)
        b.pieces = np.copy(board)
        if b.has_legal_moves(player):
            return 0
```

```
        if b.has_legal_moves(-player):
            return 0
        if b.countDiff(player) > 0:
            return 1
        return -1

    def getCanonicalForm(self, board, player):
        # return state if player==1, else return -state if player==-1
        return player*board

    def getSymmetries(self, board, pi):
        # mirror, rotational
        assert(len(pi) == self.n**2+1)  # 1 for pass
        pi_board = np.reshape(pi[:-1], (self.n, self.n))
        l = []

        for i in range(1, 5):
            for j in [True, False]:
                newB = np.rot90(board, i)
                newPi = np.rot90(pi_board, i)
                if j:
                    newB = np.fliplr(newB)
                    newPi = np.fliplr(newPi)
                l += [(newB, list(newPi.ravel()) + [pi[-1]])]
        return l

    def stringRepresentation(self, board):
        # 8x8 numpy array (canonical board)
        return board.tostring()

    def getScore(self, board, player):
        b = Board(self.n)
        b.pieces = np.copy(board)
        return b.countDiff(player)

def display(board):
    n = board.shape[0]

    for y in range(n):
        print (y,"|",end="")
    print("")
    print(" -----------------------")
    for y in range(n):
        print(y, "|",end="")    # print the row #
        for x in range(n):
            piece = board[y][x]    # get the piece to print
            if piece == -1: print("b ",end="")
            elif piece == 1: print("W ",end="")
            else:
                if x==n:
                    print("-",end="")
                else:
                    print("- ",end="")
        print("|")

    print("   -----------------------")
```

```
class OthelloNNet():
    def __init__(self, game, args):
        # game params
        self.board_x, self.board_y = game.getBoardSize()
        self.action_size = game.getActionSize()
        self.args = args

        # Renaming functions
        Relu = tf.nn.relu
        Tanh = tf.nn.tanh
        BatchNormalization = tf.layers.batch_normalization
        Dropout = tf.layers.dropout
        Dense = tf.layers.dense

        # Neural Net
        self.graph = tf.Graph()
        with self.graph.as_default():
            self.input_boards = tf.placeholder(tf.float32, shape=[None, self
            self.dropout = tf.placeholder(tf.float32)
            self.isTraining = tf.placeholder(tf.bool, name="is_training")

            x_image = tf.reshape(self.input_boards, [-1, self.board_x, self.
            h_conv1 = Relu(BatchNormalization(self.conv2d(x_image, args['num_
            h_conv2 = Relu(BatchNormalization(self.conv2d(h_conv1, args['num_
            h_conv3 = Relu(BatchNormalization(self.conv2d(h_conv2, args['num_
            h_conv4 = Relu(BatchNormalization(self.conv2d(h_conv3, args['num_
            h_conv4_flat = tf.reshape(h_conv4, [-1, args['num_channels']*(se
            s_fc1 = Dropout(Relu(BatchNormalization(Dense(h_conv4_flat, 1024
            s_fc2 = Dropout(Relu(BatchNormalization(Dense(s_fc1, 512), axis=
            self.pi = Dense(s_fc2, self.action_size)
            self.prob = tf.nn.softmax(self.pi)
            self.v = Tanh(Dense(s_fc2, 1))

            self.calculate_loss()

    def conv2d(self, x, out_channels, padding):
        return tf.layers.conv2d(x, out_channels, kernel_size=[3,3], padding=

    def calculate_loss(self):
        self.target_pis = tf.placeholder(tf.float32, shape=[None, self.actio
        self.target_vs = tf.placeholder(tf.float32, shape=[None])
        self.loss_pi =  tf.losses.softmax_cross_entropy(self.target_pis, sel
        self.loss_v = tf.losses.mean_squared_error(self.target_vs, tf.reshap
        self.total_loss = self.loss_pi + self.loss_v
        update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
        with tf.control_dependencies(update_ops):
            self.train_step = tf.train.AdamOptimizer(self.args['lr']).minimi

class Coach():
    """
    This class executes the self-play + learning. It uses the functions defi
    in Game and NeuralNet. args are specified in main.py.
    """
    def __init__(self, game, nnet, args):
        self.game = game
        self.board = game.getInitBoard()
        self.nnet = nnet
```

```python
        self.pnet = self.nnet.__class__(self.game)  # the competitor network
        self.args = args
        self.mcts = MCTS(self.game, self.nnet, self.args)

    def executeEpisode(self):
        """
        This function executes one episode of self-play, starting with playe
        As the game is played, each turn is added as a training example to
        trainExamples. The game is played till the game ends. After the game
        ends, the outcome of the game is used to assign values to each examp
        in trainExamples.

        It uses a temp=1 if episodeStep < tempThreshold, and thereafter
        uses temp=0.

        Returns:
            trainExamples: a list of examples of the form (canonicalBoard,pi
                           pi is the MCTS informed policy vector, v is +1 if
                           the player eventually won the game, else -1.
        """
        trainExamples = []
        self.board = self.game.getInitBoard()
        self.curPlayer = 1
        episodeStep = 0

        while True:
            episodeStep += 1
            canonicalBoard = self.game.getCanonicalForm(self.board,self.curP
            temp = int(episodeStep < self.args['tempThreshold'])

            pi = self.mcts.getActionProb(canonicalBoard, temp=temp)
            sym = self.game.getSymmetries(canonicalBoard, pi)
            for b,p in sym:
                trainExamples.append([b, self.curPlayer, p, None])

            action = np.random.choice(len(pi), p=pi)
            self.board, self.curPlayer = self.game.getNextState(self.board, 

            r = self.game.getGameEnded(self.board, self.curPlayer)

            if r!=0:
                return [(x[0],x[2],r*((-1)**(x[1]!=self.curPlayer))) for x i

    def learn(self):
        """
        Performs numIters iterations with numEps episodes of self-play in ea
        iteration. After every iteration, it retrains neural network with
        examples in trainExamples (which has a maximium length of maxlenofQu
        It then pits the new neural network against the old one and accepts 
        only if it wins >= updateThreshold fraction of games.
        """

        trainExamples = deque([], maxlen=self.args['maxlenOfQueue'])
        for i in range(self.args['numIters']):
            # bookkeeping
            print('------ITER ' + str(i+1) + '------')
#             end = time.time()
```

```
            for eps in range(self.args['numEps']):
                self.mcts = MCTS(self.game, self.nnet, self.args)   # reset
                trainExamples += self.executeEpisode()

                # bookkeeping + plot progress
#                 eps_time.update(time.time() - end)
#                 end = time.time()

            # training new network, keeping a copy of the old one
            self.nnet.save_checkpoint(folder=self.args['checkpoint'], filenam
            self.pnet.load_checkpoint(folder=self.args['checkpoint'], filenam
            pmcts = MCTS(self.game, self.pnet, self.args)

            self.nnet.train(trainExamples)
            nmcts = MCTS(self.game, self.nnet, self.args)

            print('PITTING AGAINST PREVIOUS VERSION')
            arena1 = Arena(lambda x: np.argmax(pmcts.getActionProb(x, temp=0
                          lambda x: np.argmax(nmcts.getActionProb(x, temp=0)
            print('before playgames')
            pwins, nwins, draws = arena1.playGames(self.args['arenaCompare']

            print('NEW/PREV WINS : %d / %d ; DRAWS : %d' % (nwins, pwins, dr
            if pwins+nwins > 0 and float(nwins)/(pwins+nwins) < self.args['u
                print('REJECTING NEW MODEL')
                self.nnet.load_checkpoint(folder=self.args['checkpoint'], fi

            else:
                print('ACCEPTING NEW MODEL')
                self.nnet.save_checkpoint(folder=self.args['checkpoint'], fi
                self.nnet.save_checkpoint(folder=self.args['checkpoint'], fi

class NNetWrapper():
    def __init__(self, game):
        self.nnet = OthelloNNet(game, args)
        self.board_x, self.board_y = game.getBoardSize()
        self.action_size = game.getActionSize()

        self.sess = tf.Session(graph=self.nnet.graph)
        self.saver = None
        with tf.Session() as temp_sess:
            temp_sess.run(tf.global_variables_initializer())
        self.sess.run(tf.variables_initializer(self.nnet.graph.get_collectio

    def train(self, examples):
        """
        examples: list of examples, each example is of form (board, pi, v)
        """

        for epoch in range(args['epochs']):
            print('EPOCH ::: ' + str(epoch+1))
#             data_time = AverageMeter()
#             batch_time = AverageMeter()
#             pi_losses = AverageMeter()
#             v_losses = AverageMeter()
            end = time.time()
```

```
            batch_idx = 0

            # self.sess.run(tf.local_variables_initializer())
            while batch_idx < int(len(examples)/args['batch_size']):
                sample_ids = np.random.randint(len(examples), size=args['bat
                boards, pis, vs = list(zip(*[examples[i] for i in sample_ids

                # predict and compute gradient and do SGD step
                input_dict = {self.nnet.input_boards: boards, self.nnet.targ

                # measure data loading time
#                 data_time.update(time.time() - end)

                # record loss
                self.sess.run(self.nnet.train_step, feed_dict=input_dict)
#                 pi_loss, v_loss = self.sess.run([self.nnet.loss_pi, self.n
#                 pi_losses.update(pi_loss, len(boards))
#                 v_losses.update(v_loss, len(boards))

                # measure elapsed time
#                 batch_time.update(time.time() - end)
                end = time.time()
                batch_idx += 1

                # plot progress


    def predict(self, board):
        """
        board: np array with board
        """
        # timing
        start = time.time()

        # preparing input
        board = board[np.newaxis, :, :]

        # run
        prob, v = self.sess.run([self.nnet.prob, self.nnet.v], feed_dict={se

        #print('PREDICTION TIME TAKEN : {0:03f}'.format(time.time()-start))
        return prob[0], v[0]

    def save_checkpoint(self, folder='checkpoint', filename='checkpoint.pth.
        filepath = os.path.join(folder, filename)
        if not os.path.exists(folder):
            print("Checkpoint Directory does not exist! Making directory {}"
            os.mkdir(folder)
        else:
            print("Checkpoint Directory exists! ")
        if self.saver == None:
            self.saver = tf.train.Saver(self.nnet.graph.get_collection('vari
        with self.nnet.graph.as_default():
            self.saver.save(self.sess, filepath)

    def load_checkpoint(self, folder='checkpoint', filename='checkpoint.pth.
        filepath = os.path.join(folder, filename)
```

```
        if not os.path.exists(filepath+'.meta'):
            raise("No model in path {}".format(filepath))
        with self.nnet.graph.as_default():
            self.saver = tf.train.Saver()
            self.saver.restore(self.sess, filepath)

if __name__=="__main__":
    g = OthelloGame(6)
    nnet = NNetWrapper(g)

    if args['load_model']:
        nnet.load_checkpoint(args['load_folder_file[0]'], args['load_folder_

    c = Coach(g, nnet, args)
    c.learn()
```

```
/Users/monilshah/anaconda3/envs/tensorflow/lib/python3.5/importlib/_boots
trap.py:222: RuntimeWarning: compiletime version 3.6 of module 'tensorflo
w.python.framework.fast_tensor_util' does not match runtime version 3.5
  return f(*args, **kwds)

WARNING:tensorflow:From /Users/monilshah/anaconda3/envs/tensorflow/lib/py
thon3.5/site-packages/tensorflow/python/ops/losses/losses_impl.py:691: so
ftmax_cross_entropy_with_logits (from tensorflow.python.ops.nn_ops) is de
precated and will be removed in a future version.
Instructions for updating:

Future major versions of TensorFlow will allow gradients to flow
into the labels input on backprop by default.

See tf.nn.softmax_cross_entropy_with_logits_v2.

------ITER 1------
Checkpoint Directory exists!
INFO:tensorflow:Restoring parameters from ./temp/temp.pth.tar
EPOCH ::: 1

-------------------------------------------------------------------------
--
KeyboardInterrupt                         Traceback (most recent call las
t)
<ipython-input-1-99e7e4c0ad5d> in <module>()
    751
    752     c = Coach(g, nnet, args)
--> 753     c.learn()

<ipython-input-1-99e7e4c0ad5d> in learn(self)
    638             pmcts = MCTS(self.game, self.pnet, self.args)
    639
--> 640             self.nnet.train(trainExamples)
    641             nmcts = MCTS(self.game, self.nnet, self.args)
    642

<ipython-input-1-99e7e4c0ad5d> in train(self, examples)
    694
    695                 # record loss
--> 696                 self.sess.run(self.nnet.train_step, feed_dict=inp
ut_dict)
```

```
    697 #                 pi_loss, v_loss = self.sess.run([self.nnet.loss
_pi, self.nnet.loss_v], feed_dict=input_dict)
    698 #                 pi_losses.update(pi_loss, len(boards))

~/anaconda3/envs/tensorflow/lib/python3.5/site-packages/tensorflow/pytho
n/client/session.py in run(self, fetches, feed_dict, options, run_metadat
a)
    893     try:
    894       result = self._run(None, fetches, feed_dict, options_ptr,
--> 895                          run_metadata_ptr)
    896       if run_metadata:
    897         proto_data = tf_session.TF_GetBuffer(run_metadata_ptr)

~/anaconda3/envs/tensorflow/lib/python3.5/site-packages/tensorflow/pytho
n/client/session.py in _run(self, handle, fetches, feed_dict, options, ru
n_metadata)
   1126     if final_fetches or final_targets or (handle and feed_dict_te
nsor):
   1127       results = self._do_run(handle, final_targets, final_fetche
s,
-> 1128                              feed_dict_tensor, options, run_metad
ata)
   1129     else:
   1130       results = []

~/anaconda3/envs/tensorflow/lib/python3.5/site-packages/tensorflow/pytho
n/client/session.py in _do_run(self, handle, target_list, fetch_list, fee
d_dict, options, run_metadata)
   1342     if handle is None:
   1343       return self._do_call(_run_fn, self._session, feeds, fetche
s, targets,
-> 1344                            options, run_metadata)
   1345     else:
   1346       return self._do_call(_prun_fn, self._session, handle, feeds
, fetches)

~/anaconda3/envs/tensorflow/lib/python3.5/site-packages/tensorflow/pytho
n/client/session.py in _do_call(self, fn, *args)
   1348   def _do_call(self, fn, *args):
   1349     try:
-> 1350       return fn(*args)
   1351     except errors.OpError as e:
   1352       message = compat.as_text(e.message)

~/anaconda3/envs/tensorflow/lib/python3.5/site-packages/tensorflow/pytho
n/client/session.py in _run_fn(session, feed_dict, fetch_list, target_lis
t, options, run_metadata)
   1327         return tf_session.TF_Run(session, options,
   1328                                  feed_dict, fetch_list, target_
list,
-> 1329                                  status, run_metadata)
   1330
   1331     def _prun_fn(session, handle, feed_dict, fetch_list):

KeyboardInterrupt:
```

```
In [ ]:
```