project3_task1

December 18, 2017

```
In [1]: import tensorflow as tf
        import numpy as np
        from boardgame import game1, game2, game3, game4, data_augmentation

        restore = True
```

1 Modified environment for the Tic-tac-toe

In order to prove that the agent can solve any game with all behaviours, wins agains the random player are not the sufficient conditions, since it gives only the probabilistic conclusion. Therefore, the functions **next_move_all** and **play_games_all** were defined, which are implementing the following:

1. During every move of the random player the amount of boards is expanded by $n_{xy} = 3 \times 3 = 9$ times. It is due to the policy of exploring every possible move on the board, which is clearly less or equal to number of cells.

2. In order to keep track of boards that where unfinished, or in other words, where random player made an invalid move to the cell which was already occupied, an array **finished_games** is used, that stores markers for all games among 9^5 if the random player is black (or 9^4 if it is white) that were finished properly: win, loss or tie.

3. For example, for *black* random player (similar strategy is helf for *white* random player) the boards look:

   - move = 1: np.repeat($[1, 2, 3, ..., 9]$, 9^4), where i represents the board where stone is on $[i \div 3, i \mod 3]$ for $[3 \times 3]$ board
   - move = 2: np.repeat($[1b_1, 2b_2, 3b_3, ..., 9b_9]$, 9^4), where ib_j represents the board where the agent put a stone is on $[b_j \div 3, b_j \mod 3]$ and the random player stone on $[i \div 3, i \mod 3]$
   - move = 3: np.repeat($[1b_11, 1b_12, ..., 1b_19, 2b_21, ..., 2b_29, ..., 9b_99]$, 9^3), where ib_jk represents the board with the stones on $[i \div 3, i \mod 3]$, $[b_j \div 3, b_j \mod 3]$, and $[j \div 3, j \mod 3]$
   - etc

```
In [2]: class my_game2(game2):

            def next_move_all(self, b, state, game_in_progress, net, rn, p, move,\
                                                    finished_games, nlevels = 1, rw = 0):
```

```
# returns next move by using neural networks
# this is a parallel version, i.e., returns next moves for multiple games
# Input arguments: self,b,state,game_in_progress,net,rn,p,move,nlevels,rw
# self: game parameters
# b: current board status for multiple games
# state: extra state for the board
# game_in_progress: 1 if game is in progress, 0 if ended
# net: neural network. can be empty (in that case 'rn' should be 1)
# rn: randomness in the move, 0: no randomness, 1: pure random
#   if rn<0, then the first |rn| moves are random
# p: current player (1: black, 2: white)
# move: k-th move (1,2,3,...)
# nlevels (optional): tree search depth (1,2, or 3). default=1
#   if nlevels is even, then 'net' should be the opponent's neural network
# rw (optional): randomization in calculating winning probabilities, default=0
# Return values
# new_board,new_state,valid_moves,wp_max,wp_all,x,y=next_move(b,
#                                                 game_in_progress,net,rn,p,move)
#   new_board: updated boards containing new moves
#   new_state: update states
#   n_valid_moves: number of valid moves
#   wp_max: best likelihood of winning
#   wp_all: likelihood of winning for all possible next moves
#   x: x coordinates of next moves
#   y: y coordinates of next moves

# board size
nx = self.nx; ny = self.ny; nxy = nx * ny
# randomness for each game & minimum r
r = rn; rmin = np.amin(r)
ng_original = b.shape[2]
# number of games
if move % 2 ==0:
    if rmin == 1:
        ng = b.shape[2] / pow(nxy, (nxy-move)//2)
    else:
        ng = b.shape[2] / pow(nxy, (nxy-1-move)//2)
else:
    if rmin == 0:
        ng = b.shape[2] / pow(nxy, (nxy-move)//2)
    else:
        ng = b.shape[2] / pow(nxy, (nxy-1-move)//2)

# proper number of different games for the current move
# slicing with appropriate stepsize
r = r[::(ng_original/ng)]
game_in_progress = game_in_progress[::(ng_original/ng)]
b = b[:,:,::(ng_original/ng)]
```

```
state = state[:,::(ng_original/ng)]
finished_games = finished_games[::(ng_original/ng)]

# number of valid moves in each game
n_valid_moves = np.zeros((ng))
# check whether moves ('nxy' moves) are valid
valid_moves = np.zeros((ng, nxy))
# win probability for each position on this each game
wp_all = np.zeros((nx, ny, ng))
# maximum over wp_all
wp_max = -np.ones((ng))
mx = np.zeros((ng))
my = np.zeros((ng))
x = -np.ones((ng))
y = -np.ones((ng))

# check nlevels
if nlevels > 3 or nlevels <= 0:
    raise Exception('# of levels not supported. Should be 1, 2, or 3.')
# total cases to consider in tree search
ncases = pow(nxy, nlevels)

# All possible boards after 'b'
d = np.zeros((nx, ny, 3, ng * ncases), dtype = np.int32)

for p1 in range(nxy):
    if rmin == 1:
        b_prev_games = b[:,:,::nxy]
        vm1, b1, state1 = self.valid(b_prev_games, state, self.xy(p1), p)
        vm_dilute = np.zeros((ng))
        vm_dilute[::nxy] = vm1
        vm_dilute = np.roll(vm_dilute, p1)
        n_valid_moves += vm_dilute
        assert np.amax(n_valid_moves) == 1

    elif rmin < 1:
        vm1, b1, state1 = self.valid(b, state, self.xy(p1), p)
        n_valid_moves += vm1
        valid_moves[:, p1] = vm1
        if nlevels == 1:
            c = 3 - p       # current player is changed to the next player
            idx = np.arange(ng) + p1 * ng
            d[:, :, 0, idx] = (b1 == c)   # current player's stone
            d[:, :, 1, idx] = (b1 == 3 - c)  # opponent's stone
            d[:, :, 2, idx] = 2 - c # 1: current player is black, 0: white
        else:
            for p2 in range(nxy):
                vm2, b2, state2 = self.valid(b1, state1, self.xy(p2), 3 - p)
```

```
                    if nlevels == 2:
                        c = p                   # current player is changed again
                        idx = np.arange((ng)) + p1 * ng + p2 * ng * nxy
                        d[:, :, 0, idx] = (b2 == c)
                        d[:, :, 1, idx] = (b2 == 3 - c)
                        d[:, :, 2, idx] = 2 - c
                    else:
                        for p3 in range(nxy):
                            vm3, b3, state3 = self.valid(b2, state2, self.xy(p3), p)
                            c = 3 - p           # current player is changed again
                            idx = np.arange(ng) + p1 * ng + p2 * ng * nxy\
                                    + p3 * ng * nxy * nxy
                            d[:, :, 0, idx] = (b3 == c)
                            d[:, :, 1, idx] = (b3 == 3 - c)
                            d[:, :, 2, idx] = 2 - c

        if rmin == 1:
            # for the random player unvalid moves are all that put
            # the stone on the occupied cell
            finished_games *= n_valid_moves
        n_valid_moves = n_valid_moves * game_in_progress

        # For operations in TensorFlow, load session and graph
        sess = tf.get_default_session()

        # Axis rollaxis for placeholder inputs
        d = np.rollaxis(d, 3)
        if rmin < 1: # if not fully random
            softout = np.zeros((d.shape[0], 3))
            size_minibatch = d.shape[0]//5 + 1
            num_batch = np.ceil(d.shape[0] / float(size_minibatch))
            for batch_index in range(int(num_batch)):
                batch_start = batch_index * size_minibatch
                batch_end = \
                        min((batch_index + 1) * size_minibatch, d.shape[0])
                indices = range(batch_start, batch_end)
                feed_dict = {'S:0': d[indices, :, :, :]}
                softout[indices, :] = sess.run(net, feed_dict = feed_dict)
            if p == 1:
                wp = 0.5 * (1 + softout[:, 1] - softout[:, 2])
            else:
                wp = 0.5 * (1 + softout[:, 2] - softout[:, 1])

            if rw != 0:
                wp = wp + np.random.rand((ng, 1)) * rw

            if nlevels >= 3:
                wp = np.reshape(wp, (ng, nxy, nxy, nxy))
```

```
            wp = np.amax(wp, axis = 3)

        if nlevels >= 2:
            wp = np.reshape(wp, (ng, nxy, nxy))
            wp = np.amin(wp, axis = 2)

        wp = np.transpose(np.reshape(wp,(nxy,ng)))
        wp = valid_moves * wp - (1 - valid_moves)
        wp_i = np.argmax(wp, axis = 1)
        mxy = self.xy(wp_i) # max position

        for p1 in range(nxy):
            pxy = self.xy(p1)
            wp_all[int(pxy[:, 0]), int(pxy[:, 1]), :] = wp[:, p1]

    new_board = np.zeros(b.shape)
    new_board[:, :, :] = b[:, :, :]
    new_state = np.zeros(state.shape)
    new_state[:, :] = state[:, :]

    for k in range(ng):
        if n_valid_moves[k]: # if there are valid moves
            if (rmin == 1):
                # if random agent do all possible actions
                # k = v*nxy + j, make jth move always same for each
                # block in [0,1,...,nxy-1]
                rxy = self.xy(k % nxy)
                isvalid, bn, sn = self.valid(b[:, :, [k]], state[:, [k]], rxy, p)
                if int(isvalid[0]):
                    #print(k//ng_prev,bn)
                    new_board[:, :, [k]] = bn
                    new_state[:, [k]] = sn
                    x[k] = rxy[0, 0]
                    y[k] = rxy[0, 1]

            else:    # act according to the value network
                isvalid, bn, sn = self.valid(b[:, :, [k]], state[:, [k]], \
                                                                mxy[[k], :], p)
                new_board[:, :, [k]] = bn
                new_state[:, [k]] = sn
                x[k] = mxy[k, 0]
                y[k] = mxy[k, 1]

        else: # if there is no valid moves
            isvalid, bn, sn = self.valid(b[:, :, [k]], state[:, [k]], \
                                                    -np.ones((1, 2)), p)
            new_state[:, [k]] = sn
```

```
    c = ng_original/ng
    return np.repeat(new_board,c, axis=2), new_state, \
            np.repeat(n_valid_moves, c, axis=0), np.repeat(wp_max,c,axis=0),\
            np.repeat(wp_all,c,axis=2), x, y, np.repeat(finished_games,c,axis=0)


def play_games_all(self, net1, r1, net2, r2, ng, max_time = 0, nargout = 1):
    # plays 'ng' games between two players
    # optional parameter: max_time (the number of moves per game), nargout (the numb
    # returns dataset and labels
    # Inputs
    # self: game parameters
    # net1: neural network playing black. can be empty (r1 should be 1 if this is em
    # r1: randomness in the move, 0: no randomness, 1: pure random
    #   if r1<0, then the first |r1| moves are random
    # net2: neural network playing white. can be empty (r2 should be 1 if this is em
    # r2: randomness in the move, 0: no randomness, 1: pure random
    #   if r2<0, then the first |r2| moves are random
    # ng: number of games to play
    # Return values
    #   stat=play_games(net1,r1,net2,r2,ng,nargout=1): statistics for net1, stat=[wi
    #   d,w,wp,stat=play_games(net1,r1,net2,r2,ng,nargout=2,3, or 4)
    #     d: 4-d matrix of size nx*ny*3*nb containing all moves, where nb is the tot
    #     w: nb*1, 0: tie, 1: black wins, 2: white wins
    #     wp (if nargout>=3):  win probabilities for the current player
    #     stat (if nargout==4): statistics for net1, stat=[win loss tie], for net2,


    # board size
    nx = self.nx; ny = self.ny
    assert np.amin(r1) == 1 or np.amin(r2) == 1, "check randomness coefficients"
    # maximum trials for each game
    if np.amin(r1) == 1:
        ng = pow(nx*ny, 5)
    elif np.amin(r2) == 1:
        ng = pow(nx*ny, 4)

    if max_time <= 0:
        np0 = nx * ny
    else:
        np0 = max_time

    # 'm' possible board configurations
    m = np0 * ng
    d = np.zeros((nx, ny, 3, m))
    pos = np.zeros((nx,ny,m))

    # Check whether tie(0)/black win(1)/white win(2) in all board configurations
```

```
w = np.zeros((m))

# winning probability: (no work for 1st generation)
wp = np.zeros((m))

# Check whether the configurations are valid for training
valid_data = np.zeros((m))

# keep track of games which contain games where random player made a move
# overlapping with existing ones
# == 0 if the game was unfinished due to invalid move from a random player
# == 1 if have not been looked at yet; or eventually was finished
finished_games = np.ones((ng))

vm0 = np.ones((ng))

if hasattr(self, 'game_init'):
    [b, state] = self.game_init(ng)
else:
    b = np.zeros((nx, ny, ng))
    state = np.zeros((0, ng))

# maximum winning probability for each game
wp_max = np.zeros((ng))

# For each time step, check whether game is in progress or not.
game_in_progress = np.ones((ng))

# First player: player 1 (black)
p = 1
for k in range(np0):
    if p == 1:
        b, state, n_valid_moves, wp_max, _, x_pos, y_pos, finished_games=\
            self.next_move_all(b, state, game_in_progress, net1, \
                                        r1, p, k, finished_games)
    else:
        b, state, n_valid_moves, wp_max, _, x_pos, y_pos, finished_games=\
            self.next_move_all(b, state, game_in_progress, net2,\
                                         r2, p, k, finished_games)

    w0, end_game, _, _ = self.winner(b, state)

    idx = np.arange(k * ng, (k + 1) * ng)
    c = 3 - p    # current player is now changed to the next player
    d[:, :, 0, idx] = (b == c)
    d[:, :, 1, idx] = (b == 3 - c)
    d[:, :, 2, idx] = 2 - c
```

```
            wp[idx] = wp_max
            valid_data[idx] = game_in_progress * (n_valid_moves > 0)

            # information about who's the current player
            game_in_progress *= (n_valid_moves > 0) * (end_game == 0)

            # if end_game==1, game ends
            # if end_game==0, game ends if no more move is possible for the current play

            number_of_games_in_progress = np.sum(game_in_progress)
            if number_of_games_in_progress == 0:
                break

            p = 3 - p
            vm0 = n_valid_moves[:]

        for k in range(np0):
            idx = np.arange(k * ng, (k + 1) * ng)
            w[idx] = w0[:] # final winner

        # player 1's stat
        valid_ng = np.sum(finished_games)
        print("valid games = %d" % valid_ng)
        win = np.sum(w0*(finished_games) == 1) / float(valid_ng)
        loss = np.sum(w0*(finished_games) == 2) / float(valid_ng)
        tie = np.sum((w0== 0)*finished_games) / float(valid_ng)

        varargout = []

        if nargout >= 2:
            fv = np.where(valid_data)[0]
            varargout.append(d[:, :, :, fv])
            varargout.append(w[fv])
            if nargout >= 3:
                varargout.append(wp[fv])
            if nargout >= 4:
                varargout.append([win, loss, tie])
        else:
            varargout.append([win, loss, tie])
        return varargout


In [3]: # Choose game tic-tac-toe
        game = my_game2()

        ######################################################################
        """                    DEFINE HYPERPARAMETERS                      """
        ######################################################################
```

```
# Initial Learning Rate
alpha = 0.001
# size of minibatch
size_minibatch = 1024
# training epoch
max_epoch = 10
# number of training steps for each generation
n_train_list = [5000, 25000, 35000, 45000]
n_test_list = [1000, 1000, 1000, 1000]
```

In [4]:
```
########################################################################
"""                 COMPUTATIONAL GRAPH CONSTRUCTION                 """
########################################################################

### DEFINE OPTIMIZER ###
def network_optimizer(Y, Y_, alpha, scope):
    # Cross entropy loss
    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits = Y, labels = Y
    # Parameters in this scope
    variables = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope = scope)
    # L2 regularization
    for i in range(len(variables)):
        loss += 0.0001 * tf.nn.l2_loss(variables[i])
    # Optimizer
    optimizer = tf.train.AdamOptimizer(alpha).minimize(loss,\
            var_list = variables)
    return loss, optimizer


### NETWORK ARCHITECTURE ###
def network(state, nx, ny):
    # Set variable initializers
    init_weight = tf.random_normal_initializer(stddev = 0.1)
    init_bias = tf.constant_initializer(0.1)

    # Create variables "weights1" and "biases1".
    weights1 = tf.get_variable("weights1", [3, 3, 3, 30], initializer = init_weight)
    biases1 = tf.get_variable("biases1", [30], initializer = init_bias)

    # Create 1st layer
    conv1 = tf.nn.conv2d(state, weights1, strides = [1, 1, 1, 1], padding = 'SAME')
    out1 = tf.nn.relu(conv1 + biases1)

    # Create variables "weights2" and "biases2".
    weights2 = tf.get_variable("weights2", [3, 3, 30, 50], initializer = init_weight)
    biases2 = tf.get_variable("biases2", [50], initializer = init_bias)

    # Create 2nd layer
```

```
        conv2 = tf.nn.conv2d(out1, weights2, strides = [1, 1, 1, 1], padding ='SAME')
        out2 = tf.nn.relu(conv2 + biases2)

        # Create variables "weights1fc" and "biases1fc".
        weights1fc = tf.get_variable("weights1fc", [nx * ny * 50, 100], \
                                                    initializer = init_weight)
        biases1fc = tf.get_variable("biases1fc", [100], initializer = init_bias)

        # Create 1st fully connected layer
        fc1 = tf.reshape(out2, [-1, nx * ny * 50])
        out1fc = tf.nn.relu(tf.matmul(fc1, weights1fc) + biases1fc)

        # Create variables "weights2fc" and "biases2fc".
        weights2fc = tf.get_variable("weights2fc", [100, 3], initializer = init_weight)
        biases2fc = tf.get_variable("biases2fc", [3], initializer = init_bias)

        # Create 2nd fully connected layer
        return tf.matmul(out1fc, weights2fc) + biases2fc


    # Input
    S = tf.placeholder(tf.float32, shape = [None, game.nx, game.ny, 3], name = "S")

    # Define network
    scope = "network"
    with tf.variable_scope(scope):
        # Estimation for unnormalized log probability
        Y = network(S, game.nx, game.ny)
        # Estimation for probability
        P = tf.nn.softmax(Y, name = "softmax")
        # Target in integer
        W = tf.placeholder(tf.int32, shape = [None], name = "W")
        # Target in one-hot vector
        Y_= tf.one_hot(W, 3, name = "Y_")
        # Define loss and optimizer for value network
        loss, optimizer = network_optimizer(Y, Y_, alpha, scope)

    ### SAVER ###
    saver = tf.train.Saver(max_to_keep = 0)
```

In [5]:
```
    ######################################################################
    """                  TRAINING AND TESTING NETWORK                  """
    ######################################################################

    with tf.Session() as sess:
        ### DEFAULT SESSION ###
        sess.as_default()
```

```
win1 = []; lose1 = []; tie1 = [];
win2 = []; lose2 = []; tie2 = [];

### VARIABLE INITIALIZATION ###
sess.run(tf.global_variables_initializer())
if not restore:
    for generation in range(len(n_train_list)):
        print("Generating training data for generation %d" % generation)
        if generation == 0:
            # number of games to play for training
            n_train = n_train_list[generation]
            # number of games to play for testing
            n_test = n_test_list[generation]
            # randomness for all games
            r1 = np.ones((n_train)) # randomness for player 1 for all games
            r2 = np.ones((n_train)) # randomness for player 2 for all games
            [d, w] = game.play_games([], r1, [], r2, n_train, nargout = 2)
        elif generation >= 1:
            # Play 'ng' games between two players using the previous
            # generation value network
            # introduce randomness in moves for robustness
            n_train = n_train_list[generation] # number of games to play for trainin
            n_test = n_test_list[generation]   # number of games to play for testing
            mt = game.nx * game.ny // 2
            r1r = np.random.rand(n_train, 1) / float(generation)
            r2r = np.random.rand(n_train, 1) / float(generation)
            r1k = np.random.randint(mt * 2, size = (n_train, 1))
            r2k = np.random.randint(mt * 2, size = (n_train, 1))
            r1 = (r1k > mt // generation) * r1r + \
                                        (r1k <= mt // generation) * (-r1k)
            r2 = (r2k > mt // generation) * r2r + \
                                        (r2k <= mt // generation) * (-r2k)
            [d, w] = game.play_games(P, r1, P, r2, n_train, nargout = 2)

        # Data augmentation
        print("Data augmentation")
        [d, w, _] = data_augmentation(d, w, [])
        d = np.rollaxis(d, 3)

        iteration = 0
        print("Start training")
        # Train the next generation value network
        for epoch in range(max_epoch):
            # random shuffling
            data_index = np.arange(len(w))
            np.random.shuffle(data_index)
            num_batch = int(np.ceil(len(data_index) / float(size_minibatch)))
            for batch_index in range(int(num_batch)):
```

```
                batch_start = batch_index * size_minibatch
                batch_end = min((batch_index + 1) * size_minibatch, len(data_index))
                indices = data_index[np.arange(batch_start, batch_end)]
                feed_dict = {S: d[indices, :, :, :], W: w[indices]}
                sess.run(optimizer, feed_dict = feed_dict)
                iteration += 1
                if iteration % 100 == 99:
                    print("Epoch: %3d\t Iteration: %6d\t Loss: %10.5f" %\
                        (epoch, iteration, sess.run(loss, feed_dict = feed_dict)))

        # Save session.
        saver.save(sess, "./project3_task1_gen" + str(generation) + ".ckpt")
        # Load session
        # saver.restore(sess, "./tictactoe_gen" + str(generation) + ".ckpt")

        print("Evaluating generation %d neural network against random policy" % gene

        r1 = np.zeros((n_test)) # randomness for player 1
        r2 = np.ones((n_test))  # randomness for player 2
        s = game.play_games(P, r1, [], r2, n_test, nargout = 1)
        win1.append(s[0][0]); lose1.append(s[0][1]); tie1.append(s[0][2]);
        print(" net plays black: win=%6.6f, loss=%6.6f, tie=%6.6f" %\
            (win1[generation], lose1[generation], tie1[generation]))

        r1 = np.ones((n_test))  # randomness for player 1
        r2 = np.zeros((n_test)) # randomness for player 2
        s = game.play_games([], r1, P, r2, n_test, nargout = 1)
        win2.append(s[0][1]); lose2.append(s[0][0]); tie2.append(s[0][2]);
        print(" net plays white: win=%6.6f, loss=%6.6f, tie=%6.6f" %\
            (win2[generation], lose2[generation], tie2[generation]))

else:
    generation = len(n_train_list)-1
    n_test = n_test_list[generation]
    saver.restore(sess, "./project3_task1_gen" + str(generation) + ".ckpt")
    print("Evaluating generation %d neural network against random policy" % generati

    r1 = np.zeros((n_test)) # randomness for player 1
    r2 = np.ones((n_test))  # randomness for player 2
    s = game.play_games(P, r1, [], r2, n_test, nargout = 1)
    print(" net plays black: win=%6.6f, loss=%6.6f, tie=%6.6f" %\
        (s[0][0],s[0][1],s[0][2]))

    r1 = np.ones((n_test))  # randomness for player 1
    r2 = np.zeros((n_test)) # randomness for player 2
    s = game.play_games([], r1, P, r2, n_test, nargout = 1)
    print(" net plays white: win=%6.6f, loss=%6.6f, tie=%6.6f" %\
        (s[0][1],s[0][0],s[0][2]))
```

```
            print("Evaluating generation %d neural network against itself" % generation)

            r1 = np.zeros((n_test)) # randomness for player 1
            r2 = np.zeros((n_test))  # randomness for player 2
            s = game.play_games(P, r1, P, r2, n_test, nargout = 1)
            print("win=%6.6f, loss=%6.6f, tie=%6.6f" %\
                (s[0][0],s[0][1],s[0][2]))
            print("\n----------------------------------\n")
            print("Evaluating generation %d neural network on all games" % generation)

            r1 = np.zeros((n_test)) # randomness for player 1
            r2 = np.ones((n_test))  # randomness for player 2
            s = game.play_games_all(P, r1, [], r2, 1, nargout = 1)
            win1.append(s[0][0]); lose1.append(s[0][1]); tie1.append(s[0][2]);
            print(" net plays black: win=%6.6f, loss=%6.6f, tie=%6.6f" %\
                 (s[0][0],s[0][1],s[0][2]))

            r1 = np.ones((n_test))  # randomness for player 1
            r2 = np.zeros((n_test)) # randomness for player 2
            s = game.play_games_all([], r1, P, r2, 1, nargout = 1)
            win2.append(s[0][1]); lose2.append(s[0][0]); tie2.append(s[0][2]);
            print(" net plays white: win=%6.6f, loss=%6.6f, tie=%6.6f" %\
                          (s[0][1],s[0][0],s[0][2]))
```

```
INFO:tensorflow:Restoring parameters from ./project3_task1_gen3.ckpt
Evaluating generation 3 neural network against random policy
 net plays black: win=0.987000, loss=0.000000, tie=0.013000
 net plays white: win=0.909000, loss=0.000000, tie=0.091000
Evaluating generation 3 neural network against itself
win=0.000000, loss=0.000000, tie=1.000000

----------------------------------

Evaluating generation 3 neural network on all games
valid games = 688
 net plays black: win=0.994186, loss=0.000000, tie=0.005814
valid games = 2373
 net plays white: win=0.963338, loss=0.000000, tie=0.036662
```

2 Analysis on the network performance

The network was trained during 5 generations, where for each generation the amount of random actions was shrinked proportionally:

```
* during the first generation the network is trained on random plays
* the second is trained based on game records played by the first neural network
```

```
* etc
```

From the results above can be seen that fully trained network never looses against the random policy. Also based on all games, network always shows the winning strategy or tie:

```
* for the black random player number of valid games is 2373 because lots of the games
repeat (e.g. board1: 134 and board2: 135 may end up in the same board like 1345,
which happens due to the agent moves as well as breadth-first search)
* for the white random player number of valid games is 688
```

And against itself the network shows the tie, hence our player is the most optimal, because it never looses and shows a tie in case it plays with another optimal player.