Name: Hong Hai Tran
Student ID: 20174583

Deep Learning with Alpha Go – Project #3

Task 1. Tic-tac-toe

- Explanation on how network is trained

  The network has 2 convolutional layers and 2 fully-connected layers. Output layer is the second fully-connected layer which has 3 output units representing 'black win', 'white win' and 'tie'.

  During the training process, 3 generations of value network is created. The 1st generation is created using training data of all random moves, i.e. from scratch. The 2nd generation using training data of both random moves and moves chosen by 1st generation of value network. The 3rd generation uses training data from both random moves and moves chosen by the 2nd generation of value network. Number of game examples for 1st, 2nd, 3rd generation of network are 5000, 30000 and 30000 respectively.

  Most of the code are taken from the sample code (tictactoe_train.py), except for the code to change number of generation and number of training examples. Epoch value is set to 10.
- Source code (project3_task1.py):
  I include some of the source code of boardgame.py in my code, including class *boardgame* and class *game2* (renamed to *tictactoe*).
  In the following, I will only show 2 things because the other parts are not changed. The two things are:
  - *next_move()* method of class *boardgame*, which I changed so that the opponent can play all possible moves
  - Evaluation part, where I let the trained value network play with the opponent mentioned above and show the result

  First part, the *next_move()* method:

```
def next_move(self, b, state, game_in_progress, net, rn, p, move, nlevels = 1, rw = 0):
    # board size
    nx = self.nx; ny = self.ny; nxy = nx * ny
    # randomness for each game & minimum r
    r = rn; rmin = np.amin(r)
    # number of games
    if b.ndim>=3:
        ng = b.shape[2]
    else:
        ng=1
    # number of valid moves in each game
    n_valid_moves = np.zeros((ng))
    # check whether each of up to 'nxy' moves is valid for each game
    valid_moves = np.zeros((ng, nxy))
    # win probability for each next move
    wp_all = np.zeros((nx, ny, ng))
    # maximum of wp_all over all possible next moves
    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)

    # maximum possible board states considering 'ncases'
    d = np.zeros((nx, ny, 3, ng * ncases), dtype = np.int32)

    for p1 in range(nxy):
        vm1, b1, state1 = self.valid(b, state, self.xy(p1), p)
        n_valid_moves += vm1
        if rmin < 1:
            valid_moves[:, p1] = vm1
            if nlevels == 1:
                c = 3 - p  # current player is changed to the next player after placing a
stone at 'p1'
                idx = np.arange(ng) + p1 * ng
                d[:, :, 0, idx] = (b1 == c)     # 1 if current player's stone is present,
0 otherwise
                d[:, :, 1, idx] = (b1 == 3 - c) # 1 if opponent's stone is present, 0 othe
rwise
                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 after plac
ing a stone at 'p2'
                        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 yet again after
placing a stone at 'p3'
                            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

    # n_valid_moves is 0 if game is not in progress
    n_valid_moves = n_valid_moves * game_in_progress

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

    # d(nx, ny, 3, ng * ncases) becomes d(ng * ncases, nx, ny, 3)
    d = np.rollaxis(d, 3)
    if rmin < 1: # if not fully random, then use the neural network 'net'
        softout = np.zeros((d.shape[0], 3))
        size_minibatch = 1024
        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, :, :, :]}  # d[indices,:,:,:] goes to 'S' (neur
al network input)
            softout[indices, :] = sess.run(net, feed_dict = feed_dict) # get softmax outpu
t from 'net'
        if p == 1:   # if the current player is black
            # softout[:,0] is the softmax output for 'tie'
            # softout[:,1] is the softmax output for 'black win'
            # softout[:,2] is the softmax output for 'white win'
            wp = 0.5 * (1 + softout[:, 1] - softout[:, 2])  # estimated win prob. for blac
k
        else:        # if the current player is white
            wp = 0.5 * (1 + softout[:, 2] - softout[:, 1])  # estimated win prob. for whit
e

        if rw != 0:     # this is only for nlevels == 1
            # add randomness so that greedy action selection to be done later is randomize
d

            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)  # greedy action selection
        mxy = self.xy(wp_i)             # convert to (x,y) coordinates

        for p1 in range(nxy):
            pxy = self.xy(p1)
            wp_all[int(pxy[:, 0]), int(pxy[:, 1]), :] = wp[:, p1]  # win prob. for each of
possible next moves

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

    # Choose action
    if rmin == 0:   # Net's turn
        for k in range(ng):
            if n_valid_moves[k]:    # If there are valid moves
                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:   # No valid moves
                isvalid, bn, sn = self.valid(b[:, :, [k]], state[:, [k]], -
np.ones((1, 2)), p)
                new_state[:, [k]] = sn
    elif rmin == 1: # Opponent's turn
        # Opponent plays black
        if move == 1:   # First move
            i = j = 0; k = 0;
            cxy = np.zeros((1,2))
            for a0 in range(9):     # Iterating over 9 possible valid moves
                if n_valid_moves[k]:    # If there are valid moves
                    while(1):   # find a possible move
                        cxy[0,:] = [i,j]
                        isvalid, _, _ = self.valid(b[:, :, [k]], state[:, [k]], cxy, p)
                        if isvalid:
                            if j != 2:
                                j += 1
                            else:
                                i += 1; j = 0;
                            break
                        else:
                            if j != 2:
                                j += 1
                            else:
                                i += 1; j = 0;
                    for a1 in range(7*5*3): # apply move cxy to 7*5*3 games
                        isvalid, bn, sn = self.valid(b[:, :, [k + a1]], state[:, [k + a1]]
, cxy, p)
                        new_board[:, :, [k + a1]] = bn
                        new_state[:, [k + a1]] = sn
                        x[k + a1] = cxy[0,0]
                        y[k + a1] = cxy[0,1]
                else:   # No valid moves
                    for a1 in range(7*5*3):
                        isvalid, bn, sn = self.valid(b[:, :, [k + a1]], state[:, [k + a1]]
, -np.ones((1, 2)), p)
                        new_state[:, [k + a1]] = sn
                k += 7*5*3
        elif move == 3:     # 3rd move
            k = 0
            cxy = np.zeros((1,2))
            for a0 in range(9):
                i = j = 0;
                for a1 in range(7): # Iterating over 7 possible valid moves
                    if n_valid_moves[k]:
                        while(1):   # Find a possible move
                            cxy[0,:] = [i,j]
                            isvalid, _, _ = self.valid(b[:, :, [k]], state[:, [k]], cxy, p
)
                            if isvalid:
                                if j != 2:
                                    j += 1
                                else:
                                    i += 1; j = 0;
                                break
                            else:
                                if j != 2:
                                    j += 1
                                else:
                                    i += 1; j = 0;
                        for a2 in range(5*3):   # Apply move cxy to 5*3 games
                            isvalid, bn, sn = self.valid(b[:, :, [k + a2]], state[:, [k +
a2]], cxy, p)
                            new_board[:, :, [k + a2]] = bn
                            new_state[:, [k + a2]] = sn
                            x[k + a2] = cxy[0,0]
                            y[k + a2] = cxy[0,1]
                    else:
                        for a2 in range(5*3):
                            isvalid, bn, sn = self.valid(b[:, :, [k + a2]], state[:, [k +
a2]], -np.ones((1, 2)), p)
                            new_state[:, [k + a2]] = sn
                    k += 5*3
        elif move == 5: # 5th move
            k = 0
            cxy = np.zeros((1,2))
            for a0 in range(9):
                for a1 in range(7):
                    i = j = 0
                    for a2 in range(5): # Iterating over 5 possible moves
                        if n_valid_moves[k]:
                            while(1):   # find a possible move
                                cxy[0,:] = [i,j]
                                isvalid, _, _ = self.valid(b[:, :, [k]], state[:, [k]], cx
y, p)
                                if isvalid:
                                    if j != 2:
                                        j += 1
                                    else:
                                        i += 1; j = 0;
                                    break
                                else:
                                    if j != 2:
                                        j += 1
                                    else:
                                        i += 1; j = 0;
                            for a3 in range(3): # Apply move cxy to 3 games
                                isvalid, bn, sn = self.valid(b[:, :, [k + a3]], state[:, [
k + a3]], cxy, p)
                                new_board[:, :, [k + a3]] = bn
                                new_state[:, [k + a3]] = sn
                                x[k + a3] = cxy[0,0]
                                y[k + a3] = cxy[0,1]
                        else:
                            for a3 in range(3):
                                isvalid, bn, sn = self.valid(b[:, :, [k + a3]], state[:, [
k + a3]], -np.ones((1, 2)), p)
                                new_state[:, [k + a3]] = sn
                        k += 3
        elif move == 7: # 7th move
            k = 0
            cxy = np.zeros((1,2))
            for a0 in range(9):
                for a1 in range(7):
                    for a2 in range(5):
                        i = j = 0
                        for a3 in range(3): # Iterate over 3 possible moves
                            if n_valid_moves[k]:
                                while(1):   # Find possible move
                                    cxy[0,:] = [i,j]
                                    isvalid, _, _ = self.valid(b[:, :, [k]], state[:, [k]]
, cxy, p)
                                    if isvalid:
                                        if j != 2:
                                            j += 1
                                        else:
                                            i += 1; j = 0;
                                        break
                                    else:
                                        if j != 2:
                                            j += 1
                                        else:
                                            i += 1; j = 0;
                                for a4 in range(1): # Apply move cxy to 1 game
                                    isvalid, bn, sn = self.valid(b[:, :, [k + a4]], state[
:, [k + a4]], cxy, p)
                                    new_board[:, :, [k + a4]] = bn
                                    new_state[:, [k + a4]] = sn
                                    x[k + a4] = cxy[0,0]
                                    y[k + a4] = cxy[0,1]
                            else:
                                for a4 in range(1):
                                    isvalid, bn, sn = self.valid(b[:, :, [k + a4]], state[
:, [k + a4]], -np.ones((1, 2)), p)
                                    new_state[:, [k + a4]] = sn
                            k += 1
        elif move == 9: # Last move (last empty cell in the 9 cells of the board game)
            for k in range(ng):
                cxy = np.zeros((1,2))
                if n_valid_moves[k]:
                    i = j = 0
                    while(1):   # Finding possible move
                        cxy[0,:] = [i,j]
                        isvalid, _, _ = self.valid(b[:, :, [k]], state[:, [k]], cxy, p)
                        if isvalid:
                            break
                        else:
                            if j != 2:
                                j += 1
                            else:
                                i += 1; j = 0;
                            continue
                    isvalid, bn, sn = self.valid(b[:, :, [k]], state[:, [k]], cxy, p)
                    new_board[:, :, [k]] = bn
                    new_state[:, [k]] = sn
                    x[k] = cxy[0,0]
                    y[k] = cxy[0,1]
                else:
                    isvalid, bn, sn = self.valid(b[:, :, [k]], state[:, [k]], -
np.ones((1, 2)), p)
                    new_state[:, [k]] = sn

        # Opponent plays white
        elif move == 2:     # 2nd move
            k = 0
            cxy = np.zeros((1,2))
            i = j = 0
            for a0 in range (8):    # Iterate over 8 possible moves
                if n_valid_moves[k]:
                    while(1):   # Finding a possible move
                        cxy[0,:] = [i,j]
                        isvalid, _, _ = self.valid(b[:, :, [k]], state[:, [k]], cxy, p)
                        if isvalid:
                            if j != 2:
                                j += 1
                            else:
                                i += 1; j = 0;
                            break
                        else:
                            if j != 2:
                                j += 1
                            else:
                                i += 1; j = 0;
                    for a1 in range(6*4*2): # apply move cxy to 6*4*2 games
                        isvalid, bn, sn = self.valid(b[:, :, [k + a1]], state[:, [k + a1]]
, cxy, p)
                        new_board[:, :, [k + a1]] = bn
                        new_state[:, [k + a1]] = sn
                        x[k + a1] = cxy[0,0]
                        y[k + a1] = cxy[0,1]
                else:
                    for a1 in range(6*4*2):
                        isvalid, bn, sn = self.valid(b[:, :, [k + a1]], state[:, [k + a1]]
, -np.ones((1, 2)), p)
                        new_state[:, [k + a1]] = sn
                k += 6*4*2
        elif move == 4:     # 4th move
            k = 0
            cxy = np.zeros((1,2))
            for a0 in range(8):
                i = j = 0;
                for a1 in range(6): # Iterate over 6 possible moves
                    if n_valid_moves[k]:
                        while(1):   # Find a possible move
                            cxy[0,:] = [i,j]
                            isvalid, _, _ = self.valid(b[:, :, [k]], state[:, [k]], cxy, p
)
                            if isvalid:
                                if j != 2:
                                    j += 1
                                else:
                                    i += 1; j = 0;
                                break
                            else:
                                if j != 2:
                                    j += 1
                                else:
                                    i += 1; j = 0;
                        for a2 in range(4*2):   # apply move cxy to 4*2 games
                            isvalid, bn, sn = self.valid(b[:, :, [k + a2]], state[:, [k +
a2]], cxy, p)
                            new_board[:, :, [k + a2]] = bn
                            new_state[:, [k + a2]] = sn
                            x[k + a2] = cxy[0,0]
                            y[k + a2] = cxy[0,1]
                    else:
                        for a2 in range(4*2):
                            isvalid, bn, sn = self.valid(b[:, :, [k + a2]], state[:, [k +
a2]], -np.ones((1, 2)), p)
                            new_state[:, [k + a2]] = sn
                    k += 4*2
        elif move == 6:     # 6th move
            k = 0
            cxy = np.zeros((1,2))
            for a0 in range(8):
                for a1 in range(6):
                    i = j = 0
                    for a2 in range(4):     # Iterate over 4 possible moves
                        if n_valid_moves[k]:
                            while(1):   # Find a possible move
```

```
                                cxy[0,:] = [i,j]
                                isvalid, _, _ = self.valid(b[:, :, [k]], state[:, [k]], cx
y, p)
                                if isvalid:
                                    if j != 2:
                                        j += 1
                                    else:
                                        i += 1; j = 0;
                                    break
                                else:
                                    if j != 2:
                                        j += 1
                                    else:
                                        i += 1; j = 0;
                            for a3 in range(2):     # Apply move cxy to 2 games
                                isvalid, bn, sn = self.valid(b[:, :, [k + a3]], state[:, [
k + a3]], cxy, p)
                                new_board[:, :, [k + a3]] = bn
                                new_state[:, [k + a3]] = sn
                                x[k + a3] = cxy[0,0]
                                y[k + a3] = cxy[0,1]
                        else:
                            for a3 in range(2):
                                isvalid, bn, sn = self.valid(b[:, :, [k + a3]], state[:, [
k + a3]], -np.ones((1, 2)), p)
                                new_state[:, [k + a3]] = sn
                        k += 2
        elif move == 8:     # Last move for white player
            k = 0
            cxy = np.zeros((1,2))
            for a0 in range(8):
                for a1 in range(6):
                    for a2 in range(4):
                        i = j = 0
                        for a3 in range(2):     # Iterate over 2 possible moves
                            if n_valid_moves[k]:
                                while(1):    # Find possible move
                                    cxy[0,:] = [i,j]
                                    isvalid, _, _ = self.valid(b[:, :, [k]], state[:, [k]]
, cxy, p)
                                    if isvalid:
                                        if j != 2:
                                            j += 1
                                        else:
                                            i += 1; j = 0;
                                        break
                                    else:
                                        if j != 2:
                                            j += 1
                                        else:
                                            i += 1; j = 0;
                                for a4 in range(1):     # apply move cxy to 1 game
                                    isvalid, bn, sn = self.valid(b[:, :, [k + a4]], state[
:, [k + a4]], cxy, p)
                                    new_board[:, :, [k + a4]] = bn
                                    new_state[:, [k + a4]] = sn
                                    x[k + a4] = cxy[0,0]
                                    y[k + a4] = cxy[0,1]
                            else:
                                for a4 in range(1):
                                    isvalid, bn, sn = self.valid(b[:, :, [k + a4]], state[
:, [k + a4]], -np.ones((1, 2)), p)
                                    new_state[:, [k + a4]] = sn
                            k += 1
        elif move == 10:
            pass
        else:
            print 'Unexpected move value. move =', move
    else:
        print 'Unexpected rmin value. rmin =', rmin

    return new_board, new_state, n_valid_moves, wp_max, wp_all, x, y
```

Secondly, the evaluation part:

```
# Declare tictactoe game
game = tictactoe()

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

# Network for loading from .ckpt
```

```
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")

saver = tf.train.Saver()

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

    ### VARIABLE INITIALIZATION ###
    sess.run(tf.global_variables_initializer())

    # Load trained parameters
    saver.restore(sess, "project3_task1.ckpt")

    # Net plays black
    ng_test = 8 * 6 * 4 * 2
    r1 = np.zeros((ng_test)) # Player 1 uses greedy actions
    r2 = np.ones((ng_test))  # Player 2 uses random actions
    s = game.play_games(P, r1, [], r2, ng_test, nargout = 1)
    win1 = s[0][0]; lose1 = s[0][1]; tie1 = s[0][2];
    print(" Net plays black: win=%6.4f, loss=%6.4f, tie=%6.4f" %\
        (win1, lose1, tie1))

    # Net plays white
    ng_test = 9 * 7 * 5 * 3
    r1 = np.ones((ng_test))  # Player 1 uses random actions
    r2 = np.zeros((ng_test)) # Player 2 uses greddy actions
    s = game.play_games([], r1, P, r2, ng_test, nargout = 1)
    win2 = s[0][1]; lose2 = s[0][0]; tie2 = s[0][2];
    print(" Net plays white: win=%6.4f, loss=%6.4f, tie=%6.4f" %\
        (win2, lose2, tie2))
```

- Game result:
```
Net plays black: win=0.9896, loss=0.0000, tie=0.0104
Net plays white: win=0.9164, loss=0.0000, tie=0.0836
```
- Explanation on code design
  - Evaluation part

    This part first defines the game, defines the network structure, then it restores the saved trained parameters and start evaluating.

    The evaluation is divided into 2 parts. The first part tests the network when it plays black. As we know, the maximum number of moves in one tic-tac-toe game is 9. The network plays black, i.e. it plays first, then the opponent, and the network, and so on. Therefore, there are 8*6*4*2 = 384 sequences in total that the opponent can play, which explains why I set number of test games in the first case to be 384.

    Similarly, in the second part, when the network plays white, there are 9*7*5*3 = 945 sequences in total that the opponent can play, which explains why the number of test games in this case is set to 945.

    In both cases, the random values are set to 0 for the network and set to 1 for the opponent, but the value of 1 does not mean the actions will be chosen randomly. It is only used to distinguish between the two players. The results of the games are printed out after playing.

- *next_move* method

  The implementation of this method is almost the same as the original one (see boardgame.py), except for the part that the action is chosen. In the original code, the actions of opponent are chosen randomly or following the network output (depending on the random values). In my code, I designed an opponent to test all possible sequences of moves.

  The code receives *move* parameter, which has values 1 to 9 to indicate the k-th move of the games. When opponent plays black, all the possible values of *move* are 1, 3, 5, 7 and 9. When *move* is 1, i.e. the first move, there are 9 possible actions that the opponents can play. However, there are 9*7*5*3 = 945 games in total, so I use one action for each 7*5*3 = 105 games. After the first move, there will be 9 different board configurations distributed equally into 945 games, i.e. each board configuration exists in 105 games. When *move* is 3, because 2 cells are occupied, there are 7 possible actions. Each board configuration (corresponding to 105 games) after the first move has a different set of 7 possible actions. For each 7*5*3 = 105 games, I apply 7 different actions equally, i.e. one action for each 5*3 = 15 games. When *move* is 5, because 4 cells are occupied, there are 5 possible actions left. At this time, there are 9*7 = 63 different board configurations. Each configuration exists in 15 games and has a different set of 5 possible actions. I apply 5 different actions equally in each 15 games, i.e. one action for each 3 games. When *move* is 7, there are 9*7*5 different board configuration boards in total, each exists in 3 games. Also at this time, 6 cells are occupied, which means there are 3 possible actions left. Each board configuration has a different set of 3 possible actions. Therefore, for each 3 games in total number of games, I apply one possible action to each game. *At this time, all the possible sequences when opponent plays black are tested*. When *move* is 9, there will be one action left in each game.

  Similarly, for the case opponent plays white. In this case, all the possible values of move are 2, 4, 6 and 8. When *move* is 2, there are 8 possible actions the opponent can take. When *move* is 4, 6 and 8, there are 6, 4 and 2 possible actions that the opponent can take. I have considered and implemented all the possible moves, as what I did in the case opponent plays black.

  The opponent played all its possible sequences of moves. The value network always wins or ties with the opponent and never loses a match, which means that it is trained properly.

<u>Task 2.</u> Mini AlphaGo Zero

- Explanation on how network is trained

  The network has 3 convolutional layers and 2 fully-connected layers, i.e. one more convolutional layer than the network in task 1. Output layer is the second fully-connected layer which has 3 output units representing 'black win', 'white win' and 'tie'.

  During the training process, 5 generations of value network is created. The 1st generation is created using training data of all random moves, i.e. from scratch. The 2nd generation using training data of both random moves and moves chosen by 1st generation of value network, the 3rd generation uses training data from both random moves and moves chosen by the 2nd generation of value network, and so on. Number of game examples for generations of network are 2000, 5000, 10000, 15000 and 20000 respectively. The epoch number is set to 25.

  Most of the code are taken from the sample code (go_train.py).
- Source code (project3_task2.py)

```
import tensorflow as tf
import numpy as np
from boardgame import game1, game2, game3, game4, data_augmentation

# Choose game Go
game = game1()

### 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 "weights3" and "biases3".
    weights3 = tf.get_variable("weights3", [3, 3, 50, 70], initializer = init_weight)
    biases3 = tf.get_variable("biases3", [70], initializer = init_bias)

    # Create 3rd layer
    conv3 = tf.nn.conv2d(out2, weights3, strides = [1, 1, 1, 1], padding ='SAME')
    out3 = tf.nn.relu(conv3 + biases3)

    # Create variables "weights1fc" and "biases1fc".
    weights1fc = tf.get_variable("weights1fc", [nx * ny * 70, 100], initializer = init_wei
ght)
    biases1fc = tf.get_variable("biases1fc", [100], initializer = init_bias)
    # Create 1st fully connected layer
    fc1 = tf.reshape(out3, [-1, nx * ny * 70])
    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 (common for all networks)
S = tf.placeholder(tf.float32, shape = [None, game.nx, game.ny, 3], name = "S")

# temporary network for loading from .ckpt
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")

for i in range(5):
    scope = "network" + str(i)
    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")

N_variables = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope = "network/")
N0_variables = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope = "network0/")
N1_variables = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope = "network1/")
N2_variables = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope = "network2/")
N3_variables = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope = "network3/")
N4_variables = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope = "network4/")

### SAVER ###
saver = tf.train.Saver(N_variables)

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

    ### VARIABLE INITIALIZATION ###
    sess.run(tf.global_variables_initializer())
    n_test = 1
    r_none = np.zeros((n_test))
    # Load trained params for net 1
    saver.restore(sess, "./project3_task2_gen0.ckpt")
    for i in range(len(N_variables)):
        sess.run(tf.assign(N0_variables[i], N_variables[i]))
    # Load trained params for net 2
    saver.restore(sess, "./project3_task2_gen1.ckpt")
    for i in range(len(N_variables)):
        sess.run(tf.assign(N1_variables[i], N_variables[i]))
    # Load trained params for net 3
    saver.restore(sess, "./project3_task2_gen2.ckpt")
    for i in range(len(N_variables)):
        sess.run(tf.assign(N2_variables[i], N_variables[i]))
    # Load trained params for net 4
    saver.restore(sess, "./project3_task2_gen3.ckpt")
    for i in range(len(N_variables)):
        sess.run(tf.assign(N3_variables[i], N_variables[i]))
    # Load trained params for net 5
    saver.restore(sess, "./project3_task2_gen4.ckpt")
    for i in range(len(N_variables)):
        sess.run(tf.assign(N4_variables[i], N_variables[i]))

    N0 = tf.get_default_graph().get_tensor_by_name("network0/softmax:0")
    N1 = tf.get_default_graph().get_tensor_by_name("network1/softmax:0")
    N2 = tf.get_default_graph().get_tensor_by_name("network2/softmax:0")
    N3 = tf.get_default_graph().get_tensor_by_name("network3/softmax:0")
    N4 = tf.get_default_graph().get_tensor_by_name("network4/softmax:0")

    s = game.play_games(N0, r_none, N0, r_none, n_test, nargout = 1)
    win=s[0][0]; loss=s[0][1]; tie=s[0][2]
    print('net1 (black) against net1 (white): win %d, loss %d, tie %d' % (win, loss, tie))

    s = game.play_games(N0, r_none, N1, r_none, n_test, nargout = 1)
    win=s[0][0]; loss=s[0][1]; tie=s[0][2]
    print('net1 (black) against net2 (white): win %d, loss %d, tie %d' % (win, loss, tie))

    s = game.play_games(N0, r_none, N2, r_none, n_test, nargout = 1)
    win=s[0][0]; loss=s[0][1]; tie=s[0][2]
    print('net1 (black) against net3 (white): win %d, loss %d, tie %d' % (win, loss, tie))

    s = game.play_games(N0, r_none, N3, r_none, n_test, nargout = 1)
    win=s[0][0]; loss=s[0][1]; tie=s[0][2]
    print('net1 (black) against net4 (white): win %d, loss %d, tie %d' % (win, loss, tie))

    s = game.play_games(N0, r_none, N4, r_none, n_test, nargout = 1)
    win=s[0][0]; loss=s[0][1]; tie=s[0][2]
    print('net1 (black) against net5 (white): win %d, loss %d, tie %d' % (win, loss, tie))


    s = game.play_games(N1, r_none, N0, r_none, n_test, nargout = 1)
    win=s[0][0]; loss=s[0][1]; tie=s[0][2]
    print('net2 (black) against net1 (white): win %d, loss %d, tie %d' % (win, loss, tie))

    s = game.play_games(N1, r_none, N1, r_none, n_test, nargout = 1)
    win=s[0][0]; loss=s[0][1]; tie=s[0][2]
    print('net2 (black) against net2 (white): win %d, loss %d, tie %d' % (win, loss, tie))

    s = game.play_games(N1, r_none, N2, r_none, n_test, nargout = 1)
    win=s[0][0]; loss=s[0][1]; tie=s[0][2]
    print('net2 (black) against net3 (white): win %d, loss %d, tie %d' % (win, loss, tie))

    s = game.play_games(N1, r_none, N3, r_none, n_test, nargout = 1)
    win=s[0][0]; loss=s[0][1]; tie=s[0][2]
    print('net2 (black) against net4 (white): win %d, loss %d, tie %d' % (win, loss, tie))

    s = game.play_games(N1, r_none, N4, r_none, n_test, nargout = 1)
    win=s[0][0]; loss=s[0][1]; tie=s[0][2]
    print('net2 (black) against net5 (white): win %d, loss %d, tie %d' % (win, loss, tie))


    s = game.play_games(N2, r_none, N0, r_none, n_test, nargout = 1)
    win=s[0][0]; loss=s[0][1]; tie=s[0][2]
    print('net3 (black) against net1 (white): win %d, loss %d, tie %d' % (win, loss, tie))

    s = game.play_games(N2, r_none, N1, r_none, n_test, nargout = 1)
    win=s[0][0]; loss=s[0][1]; tie=s[0][2]
    print('net3 (black) against net2 (white): win %d, loss %d, tie %d' % (win, loss, tie))

    s = game.play_games(N2, r_none, N2, r_none, n_test, nargout = 1)
    win=s[0][0]; loss=s[0][1]; tie=s[0][2]
    print('net3 (black) against net3 (white): win %d, loss %d, tie %d' % (win, loss, tie))

    s = game.play_games(N2, r_none, N3, r_none, n_test, nargout = 1)
    win=s[0][0]; loss=s[0][1]; tie=s[0][2]
    print('net3 (black) against net4 (white): win %d, loss %d, tie %d' % (win, loss, tie))

    s = game.play_games(N2, r_none, N4, r_none, n_test, nargout = 1)
    win=s[0][0]; loss=s[0][1]; tie=s[0][2]
    print('net3 (black) against net5 (white): win %d, loss %d, tie %d' % (win, loss, tie))


    s = game.play_games(N3, r_none, N0, r_none, n_test, nargout = 1)
    win=s[0][0]; loss=s[0][1]; tie=s[0][2]
    print('net4 (black) against net1 (white): win %d, loss %d, tie %d' % (win, loss, tie))

    s = game.play_games(N3, r_none, N1, r_none, n_test, nargout = 1)
    win=s[0][0]; loss=s[0][1]; tie=s[0][2]
    print('net4 (black) against net2 (white): win %d, loss %d, tie %d' % (win, loss, tie))

    s = game.play_games(N3, r_none, N2, r_none, n_test, nargout = 1)
    win=s[0][0]; loss=s[0][1]; tie=s[0][2]
    print('net4 (black) against net3 (white): win %d, loss %d, tie %d' % (win, loss, tie))

    s = game.play_games(N3, r_none, N3, r_none, n_test, nargout = 1)
    win=s[0][0]; loss=s[0][1]; tie=s[0][2]
    print('net4 (black) against net4 (white): win %d, loss %d, tie %d' % (win, loss, tie))

    s = game.play_games(N3, r_none, N4, r_none, n_test, nargout = 1)
    win=s[0][0]; loss=s[0][1]; tie=s[0][2]
    print('net4 (black) against net5 (white): win %d, loss %d, tie %d' % (win, loss, tie))


    s = game.play_games(N4, r_none, N0, r_none, n_test, nargout = 1)
    win=s[0][0]; loss=s[0][1]; tie=s[0][2]
    print('net5 (black) against net1 (white): win %d, loss %d, tie %d' % (win, loss, tie))

    s = game.play_games(N4, r_none, N1, r_none, n_test, nargout = 1)
    win=s[0][0]; loss=s[0][1]; tie=s[0][2]
    print('net5 (black) against net2 (white): win %d, loss %d, tie %d' % (win, loss, tie))

    s = game.play_games(N4, r_none, N2, r_none, n_test, nargout = 1)
    win=s[0][0]; loss=s[0][1]; tie=s[0][2]
    print('net5 (black) against net3 (white): win %d, loss %d, tie %d' % (win, loss, tie))

    s = game.play_games(N4, r_none, N3, r_none, n_test, nargout = 1)
    win=s[0][0]; loss=s[0][1]; tie=s[0][2]
    print('net5 (black) against net4 (white): win %d, loss %d, tie %d' % (win, loss, tie))

    s = game.play_games(N4, r_none, N4, r_none, n_test, nargout = 1)
    win=s[0][0]; loss=s[0][1]; tie=s[0][2]
    print('net5 (black) against net5 (white): win %d, loss %d, tie %d' % (win, loss, tie))
```

- Game result
```
net1 (black) against net1 (white): win 1, loss 0, tie 0
net1 (black) against net2 (white): win 0, loss 1, tie 0
net1 (black) against net3 (white): win 0, loss 1, tie 0
net1 (black) against net4 (white): win 0, loss 1, tie 0
net1 (black) against net5 (white): win 0, loss 1, tie 0

net2 (black) against net1 (white): win 1, loss 0, tie 0
net2 (black) against net2 (white): win 1, loss 0, tie 0
net2 (black) against net3 (white): win 0, loss 0, tie 1
net2 (black) against net4 (white): win 0, loss 1, tie 0
net2 (black) against net5 (white): win 0, loss 0, tie 1

net3 (black) against net1 (white): win 1, loss 0, tie 0
net3 (black) against net2 (white): win 1, loss 0, tie 0
net3 (black) against net3 (white): win 1, loss 0, tie 0
net3 (black) against net4 (white): win 0, loss 0, tie 1
net3 (black) against net5 (white): win 1, loss 0, tie 0

net4 (black) against net1 (white): win 1, loss 0, tie 0
net4 (black) against net2 (white): win 1, loss 0, tie 0
net4 (black) against net3 (white): win 1, loss 0, tie 0
net4 (black) against net4 (white): win 1, loss 0, tie 0
net4 (black) against net5 (white): win 0, loss 0, tie 1

net5 (black) against net1 (white): win 1, loss 0, tie 0
net5 (black) against net2 (white): win 1, loss 0, tie 0
net5 (black) against net3 (white): win 1, loss 0, tie 0
net5 (black) against net4 (white): win 1, loss 0, tie 0
net5 (black) against net5 (white): win 1, loss 0, tie 0
```

- Explanation on code design

  First the Go game is declared. After that, I define and declare 5 different networks for 5 different generations. Then, 5 different sets of trained parameters are restored and loaded into the 5 corresponding networks. Finally, I let the networks play against each other one by one (there are 25 games in total) and print out the results of the games. Looking at the result, we can see that the 5th generation of network has the best performance.