project3_task2

December 18, 2017

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

        # Choose game Go
        game = game1()
```

1 Training 5 agents

```
In [ ]: ######################################################################
        """                       TRAINING 5X5 GO AGENT                      """
        ######################################################################

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

        # Choose game Go
        game = game1()

        ######################################################################
        """                       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 = [3000, 10000, 18000, 25000, 45000]
        n_test_list = [1000, 1000, 1000, 1000, 1000]

        ######################################################################
        """                   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 "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_w
    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
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)

####################################################################
"""                   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())
```

```
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)
    else:
        # 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 training
        n_test = n_test_list[generation]   # number of games to play for testing
        mt = game.nx * game.ny // 2
        # decrease randomness with generations
        r1r = np.random.rand(n_train, 1) / float(np.clip(generation, 0, 4))
        r2r = np.random.rand(n_train, 1) / float(np.clip(generation, 0, 4))
        r1k = np.random.randint(mt * 2, size = (n_train, 1))
        r2k = np.random.randint(mt * 2, size = (n_train, 1))
        r1 = (r1k > mt // np.clip(generation, 0, 4)) * r1r + \
                            (r1k <= mt // np.clip(generation, 0, 4)) * (-r1k
        r2 = (r2k > mt // np.clip(generation, 0, 4)) * r2r + \
                            (r2k <= mt // np.clip(generation, 0, 4)) * (-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 = 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_task2_gen" + str(generation) + ".ckpt")
        # Load session
        # saver.restore(sess, "./go_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)
        win1.append(s[0][0]); lose1.append(s[0][1]); tie1.append(s[0][2]);
        print(" net plays black: win=%6.4f, loss=%6.4f, tie=%6.4f" %\
            (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=%6.4f, loss=%6.4f, tie=%6.4f" %\
            (win2[generation], lose2[generation], tie2[generation]))
```

2 25 round-robin

In [2]:
```
### 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_w
        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")
```

```
In [3]: # network0 for black
        # network1 for white
        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/")
        nets_vars = [N0_variables,N1_variables,N2_variables,N3_variables,N4_variables]
        ### SAVER ###
        saver = tf.train.Saver(N_variables)
```

```
In [4]: 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))
            nets = []
            for n in range(len(nets_vars)):
                saver.restore(sess, "./project3_task2_gen%d.ckpt"%(n))
                for i in range(len(N_variables)):
                    sess.run(tf.assign(nets_vars[n][i], N_variables[i]))
                nets += [tf.get_default_graph().get_tensor_by_name("network%d/softmax:0"%(n))]

            for i in range(len(nets)):
                for j in range(len(nets)):
                    s = game.play_games(nets[i], r_none, nets[j], r_none, n_test, nargout = 1)
                    win=s[0][0]; loss=s[0][1]; tie=s[0][2]
                    print('net%d (black) against net%d (white): win %d, loss %d, tie %d' % \
                                                                          (i+1, j+1, win, loss, tie))
```

```
INFO:tensorflow:Restoring parameters from ./project3_task2_gen0.ckpt
INFO:tensorflow:Restoring parameters from ./project3_task2_gen1.ckpt
INFO:tensorflow:Restoring parameters from ./project3_task2_gen2.ckpt
INFO:tensorflow:Restoring parameters from ./project3_task2_gen3.ckpt
INFO:tensorflow:Restoring parameters from ./project3_task2_gen4.ckpt
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 1, tie 0
```

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

3 Explanations

For training the following settings were used: 1. n_train_list = [3000, 10000, 18000, 25000, 45000] 2. n_test_list = [1000, 1000, 1000, 1000, 1000] 3. Exploration was decreasing proportionaly to the generation number,as shown in the code below

From the 25 round-robin results above it can be seen that: 1. Black player has a tendecy to win more frequently than the white player: * whenever the agent with the same parameters and from the same generation plays against itself, the white player always looses 2. With more generations the agent becomes more powerful, because it sees more games and the value function becomes more accurate. In this simplified setting only the value function is used without the Monte Carlo Tree Search for policy optimization, hence there is no policy evaluation and improvement loop, and the single loss signal is coming at the end of the game.

3. Due to simplifiaction in the agent network explained above and simple exploration policy (without visit count), the agent from 5th generation white agent looses against the 4th generation black agent; as well as 4th generation white agent could not beat the 3rd generation black agent. Therefore in order to make the agent more effective the algorithm has to be improved itself.