```
import time
import numpy as np
#import tensorflow as tf
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

from models import GAT
from utils import process
import time
import pickle
import argparse

import maddpg.common.tf_util as U
from maddpg.trainer.maddpg import MADDPGAgentTrainer
import tf_slim as layers


#maddpg
#Set default arguments to run MADDPG
def parse_args():
    parser = argparse.ArgumentParser("Reinforcement Learning experiments for multiagent environments")
    # Environment
    parser.add_argument("--scenario", type=str, default="simple", help="name of the scenario script")
    parser.add_argument("--max-episode-len", type=int, default=25, help="maximum episode length")
    parser.add_argument("--num-episodes", type=int, default=60000, help="number of episodes")
    parser.add_argument("--num-adversaries", type=int, default=0, help="number of adversaries")
    parser.add_argument("--good-policy", type=str, default="maddpg", help="policy for good agents")
    parser.add_argument("--adv-policy", type=str, default="maddpg", help="policy of adversaries")
    # Core training parameters
    parser.add_argument("--lr", type=float, default=1e-2, help="learning rate for Adam optimizer")
    parser.add_argument("--gamma", type=float, default=0.95, help="discount factor")
    parser.add_argument("--batch-size", type=int, default=1024, help="number of episodes to optimize at the same
time")
    parser.add_argument("--num-units", type=int, default=64, help="number of units in the mlp")
    # Checkpointing
    parser.add_argument("--exp-name", type=str, default=None, help="name of the experiment")
    parser.add_argument("--save-dir", type=str, default="/tmp/policy/", help="directory in which training state and model should be saved")
    parser.add_argument("--save-rate", type=int, default=1000, help="save model once every time this many episodes are completed")
    parser.add_argument("--load-dir", type=str, default="", help="directory in which training state and model are loaded")
    # Evaluation
    parser.add_argument("--restore", action="store_true", default=False)
    parser.add_argument("--display", action="store_true", default=False)
    parser.add_argument("--benchmark", action="store_true", default=False)
    parser.add_argument("--benchmark-iters", type=int, default=100000, help="number of iterations run for benchmarking")
    parser.add_argument("--benchmark-dir", type=str, default="./benchmark_files/", help="directory where benchmark data is saved")
    parser.add_argument("--plots-dir", type=str, default="./learning_curves/", help="directory where plot data is saved")
    return parser.parse_args()
# make environment for now. Once env is made you can bypass this step
def make_env(scenario_name, arglist, benchmark=False):
    from multiagent.environment import MultiAgentEnv
    import multiagent.scenarios as scenarios

    # load scenario from script
    scenario = scenarios.load(scenario_name + ".py").Scenario()
    # create world
    world = scenario.make_world()
    # create multiagent environment
    if benchmark:
        env = MultiAgentEnv(world, scenario.reset_world, scenario.reward, scenario.observation, scenario.benchmark_data)
    else:
        env = MultiAgentEnv(world, scenario.reset_world, scenario.reward, scenario.observation)
    return env

def mlp_model(input, num_outputs, scope, reuse=False, num_units=64, rnn_cell=None):
    # This model takes as input an observation and returns values of all actions
    with tf.variable_scope(scope, reuse=reuse):
        out = input
        out = layers.fully_connected(out, num_outputs=num_units, activation_fn=tf.nn.relu)
        out = layers.fully_connected(out, num_outputs=num_units, activation_fn=tf.nn.relu)
        out = layers.fully_connected(out, num_outputs=num_outputs, activation_fn=None)
        return out
#define multiple agents for MADDPG
def get_trainers(env, num_adversaries, obs_shape_n, arglist):
    trainers = []
    model = mlp_model
    trainer = MADDPGAgentTrainer
    for i in range(num_adversaries):
        trainers.append(trainer(
            "agent_%d" % i, model, obs_shape_n, env.action_space, i, arglist,
            local_q_func=(arglist.adv_policy=='ddpg')))
    for i in range(num_adversaries, env.n):
        trainers.append(trainer(
            "agent_%d" % i, model, obs_shape_n, env.action_space, i, arglist,
            local_q_func=(arglist.good_policy=='ddpg')))
    return trainers

#GAT first then feed it into MADDPG
def train(arglist):
    checkpt_file = 'pre_trained/cora/mod_cora.ckpt'

    dataset = 'cora'
    # training params
    batch_size = 1
    nb_epochs = 100000
    patience = 100
    lr = 0.005  # learning rate
    l2_coef = 0.0005  # weight decay
    hid_units = [8] # numbers of hidden units per each attention head in each layer
    n_heads = [8, 1] # additional entry for the output layer
    residual = False
    nonlinearity = tf.nn.elu
    model = GAT

    print('Dataset: ' + dataset)
    print('----- Opt. hyperparams -----')
    print('lr: ' + str(lr))
    print('l2_coef: ' + str(l2_coef))
    print('----- Archi. hyperparams -----')
    print('nb. layers: ' + str(len(hid_units)))
    print('nb. units per layer: ' + str(hid_units))
    print('nb. attention heads: ' + str(n_heads))
    print('residual: ' + str(residual))
    print('nonlinearity: ' + str(nonlinearity))
    print('model: ' + str(model))

    #GAT Variables
    adj, features, y_train, y_val, y_test, train_mask, val_mask, test_mask = process.load_data(dataset)
    features, spars = process.preprocess_features(features)

    nb_nodes = features.shape[0]
    ft_size = features.shape[1]
    nb_classes = y_train.shape[1]

    adj = adj.todense()

    features = features[np.newaxis]
    adj = adj[np.newaxis]
    y_train = y_train[np.newaxis]
    y_val = y_val[np.newaxis]
    y_test = y_test[np.newaxis]
    train_mask = train_mask[np.newaxis]
    val_mask = val_mask[np.newaxis]
    test_mask = test_mask[np.newaxis]
    #there is only one layer, with 8 nodes.
    biases = process.adj_to_bias(adj, [nb_nodes], nhood=1)

    #maddpg variables most of them are for diagnostics
    trainers = get_trainers(env, num_adversaries, obs_shape_n, arglist)
    episode_rewards = [0.0]  # sum of rewards for all agents
    agent_rewards = [[0.0] for _ in range(env.n)]  # individual agent reward
    final_ep_rewards = []  # sum of rewards for training curve
```

```
final_ep_ag_rewards = []  # agent rewards for training curve
agent_info = [[[]]]  # placeholder for benchmarking info
saver = tf.train.Saver()
obs_n = env.reset()
episode_step = 0
train_step = 0
training_len = 1000
t_start = time.time()

#Set Graph as the default action space
with tf.Graph().as_default():
    with tf.name_scope('input'):
        ftr_in = tf.placeholder(dtype=tf.float32, shape=(batch_size, nb_nodes, ft_size))
        bias_in = tf.placeholder(dtype=tf.float32, shape=(batch_size, nb_nodes, nb_nodes))
        lbl_in = tf.placeholder(dtype=tf.int32, shape=(batch_size, nb_nodes, nb_classes))
        msk_in = tf.placeholder(dtype=tf.int32, shape=(batch_size, nb_nodes))
        attn_drop = tf.placeholder(dtype=tf.float32, shape=())
        ffd_drop = tf.placeholder(dtype=tf.float32, shape=())
        is_train = tf.placeholder(dtype=tf.bool, shape=())

    # logits --- output of the GAT model. Its one dimensional.
    logits = model.inference(ftr_in, nb_classes, nb_nodes, is_train,
                             attn_drop, ffd_drop,
                             bias_mat=bias_in,
                             hid_units=hid_units, n_heads=n_heads,
                             residual=residual, activation=nonlinearity)
    log_resh = tf.reshape(logits, [-1, nb_classes])
    lab_resh = tf.reshape(lbl_in, [-1, nb_classes])
    msk_resh = tf.reshape(msk_in, [-1])
    loss = model.masked_softmax_cross_entropy(log_resh, lab_resh, msk_resh)
    accuracy = model.masked_accuracy(log_resh, lab_resh, msk_resh)

    train_op = model.training(loss, lr, l2_coef)

    #maddpg
    # once env is set up, make both environments the same
    env = make_env("simple", arglist, benchmark=False)
    # change the dimension of output to 1, size of the output from GAT
    obs_shape_n = [[1] for i in range(env.n)]
    num_adversaries = 0

    while train_step < training_len:
        # get action, replace obs_n with logits (output of logits). All the agents will be working with the same observation
        action_n = [agent.action(obs) for agent, obs in zip(trainers,logits)]
        # environment step
        new_obs_n, rew_n, done_n, info_n = env.step(action_n)
        episode_step += 1
        done = all(done_n)
        terminal = (episode_step >= arglist.max_episode_len)

        # collect experience
        for i, agent in enumerate(trainers):
            agent.experience(obs_n[i], action_n[i], rew_n[i], new_obs_n[i], done_n[i], terminal)
        obs_n = new_obs_n

        train_step += 1
        # update all trainers, if not in display or benchmark mode
        loss = None
        for agent in trainers:
            agent.preupdate()
        for agent in trainers:
            loss = agent.update(trainers, train_step)

        # set up diagnostics later.

    # diagnostics for GAT
    init_op = tf.group(tf.global_variables_initializer(), tf.local_variables_initializer())

    vlss_mn = np.inf
    vacc_mx = 0.0
    curr_step = 0

    with tf.Session() as sess:
```

```
sess.run(init_op)

train_loss_avg = 0
train_acc_avg = 0
val_loss_avg = 0
val_acc_avg = 0

for epoch in range(nb_epochs):
  tr_step = 0
  tr_size = features.shape[0]

  while tr_step * batch_size < tr_size:
    _, loss_value_tr, acc_tr = sess.run([train_op, loss, accuracy],
      feed_dict={
        ftr_in: features[tr_step*batch_size:(tr_step+1)*batch_size],
        bias_in: biases[tr_step*batch_size:(tr_step+1)*batch_size],
        lbl_in: y_train[tr_step*batch_size:(tr_step+1)*batch_size],
        msk_in: train_mask[tr_step*batch_size:(tr_step+1)*batch_size],
        is_train: True,
        attn_drop: 0.6, ffd_drop: 0.6})
    train_loss_avg += loss_value_tr
    train_acc_avg += acc_tr
    tr_step += 1

  vl_step = 0
  vl_size = features.shape[0]

  while vl_step * batch_size < vl_size:
    loss_value_vl, acc_vl = sess.run([loss, accuracy],
      feed_dict={
        ftr_in: features[vl_step*batch_size:(vl_step+1)*batch_size],
        bias_in: biases[vl_step*batch_size:(vl_step+1)*batch_size],
        lbl_in: y_val[vl_step*batch_size:(vl_step+1)*batch_size],
        msk_in: val_mask[vl_step*batch_size:(vl_step+1)*batch_size],
        is_train: False,
        attn_drop: 0.0, ffd_drop: 0.0})
    val_loss_avg += loss_value_vl
```

```
            val_acc_avg += acc_vl
            vl_step += 1

        print('Training: loss = %.5f, acc = %.5f | Val: loss = %.5f, acc = %.5f' %
                (train_loss_avg/tr_step, train_acc_avg/tr_step,
                val_loss_avg/vl_step, val_acc_avg/vl_step))

        if val_acc_avg/vl_step >= vacc_mx or val_loss_avg/vl_step <= vlss_mn:
            if val_acc_avg/vl_step >= vacc_mx and val_loss_avg/vl_step <= vlss_mn:
                vacc_early_model = val_acc_avg/vl_step
                vlss_early_model = val_loss_avg/vl_step
                saver.save(sess, checkpt_file)
            vacc_mx = np.max((val_acc_avg/vl_step, vacc_mx))
            vlss_mn = np.min((val_loss_avg/vl_step, vlss_mn))
            curr_step = 0
        else:
            curr_step += 1
            if curr_step == patience:
                print('Early stop! Min loss: ', vlss_mn, ', Max accuracy: ', vacc_mx)
                print('Early stop model validation loss: ', vlss_early_model, ', accuracy: ', vacc_early_model)
                break

        train_loss_avg = 0
        train_acc_avg = 0
        val_loss_avg = 0
        val_acc_avg = 0

    saver.restore(sess, checkpt_file)

    ts_size = features.shape[0]
    ts_step = 0
    ts_loss = 0.0
    ts_acc = 0.0

    while ts_step * batch_size < ts_size:
        loss_value_ts, acc_ts = sess.run([loss, accuracy],
            feed_dict={
```

```
                        ftr_in: features[ts_step*batch_size:(ts_step+1)*batch_size],
                        bias_in: biases[ts_step*batch_size:(ts_step+1)*batch_size],
                        lbl_in: y_test[ts_step*batch_size:(ts_step+1)*batch_size],
                        msk_in: test_mask[ts_step*batch_size:(ts_step+1)*batch_size],
                        is_train: False,
                        attn_drop: 0.0, ffd_drop: 0.0})
                ts_loss += loss_value_ts
                ts_acc += acc_ts
                ts_step += 1

            print('Test loss:', ts_loss/ts_step, '; Test accuracy:', ts_acc/ts_step)

            sess.close()

if __name__ == '__main__':
    arglist = parse_args()
    train(arglist)
```