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)