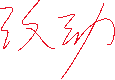
from \_\_future\_\_ import print\_function



import random

import numpy as np

import tensorflow as tf

import gc

import sys

from replay\_buffer import ReplayBuffer

from time import gmtime, strftime

logf = open("my\_net22/log\_" + strftime("%Y-%m-%d-%H-%M-%S", gmtime()) + ".txt", 'a+')

loss\_out\_path = "my\_net22/loss\_" + strftime("%Y-%m-%d-%H-%M-%S", gmtime()) + ".txt"

rewards\_out\_path = "my\_net22/reward\_out\_" + strftime("%Y-%m-%d-%H-%M-%S", gmtime()) + ".txt"

class NeuralQLearner(object):

def \_\_init\_\_(self, session,

optimizer,



q\_network,



restore\_net\_path,



state\_dim,

num\_actions,

batch\_size,



init\_exp, # initial exploration prob

final\_exp, # final exploration prob

anneal\_steps, # N steps for annealing exploration

replay\_buffer\_size,

store\_replay\_every, # how frequent to store experience

discount\_factor, # discount future rewards

target\_update\_rate,

reg\_param, # regularization constants

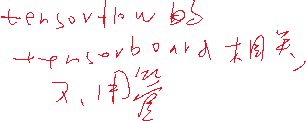
max\_gradient, # max gradient norms

double\_q\_learning,

summary\_writer,



summary\_every):



# tensorflow machinery

self.session = session

self.optimizer = optimizer

self.summary\_writer = summary\_writer

# model components

self.q\_network = q\_network

self.restore\_net\_path = restore\_net\_path

self.replay\_buffer = ReplayBuffer(buffer\_size=replay\_buffer\_size)

# Q learning parameters

self.batch\_size = batch\_size

self.state\_dim = state\_dim

self.num\_actions = num\_actions

self.exploration = init\_exp

self.init\_exp = init\_exp

self.final\_exp = final\_exp

self.anneal\_steps = anneal\_steps

self.discount\_factor = discount\_factor

self.target\_update\_rate = target\_update\_rate

self.double\_q\_learning = double\_q\_learning

# training parameters

self.max\_gradient = max\_gradient

self.reg\_param = reg\_param

# counters

self.store\_replay\_every = store\_replay\_every

self.store\_experience\_cnt = 0

self.train\_iteration = 0

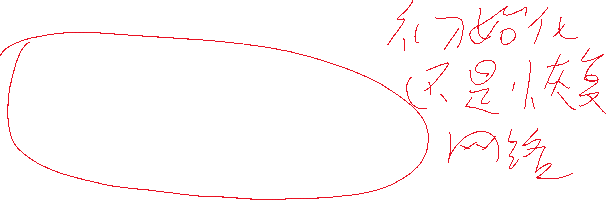
# create and initialize variables

self.create\_variables()

if self.restore\_net\_path is not None:

saver = tf.train.Saver()

saver.restore(self.session, self.restore\_net\_path)



else:

var\_lists = tf.get\_collection(tf.GraphKeys.VARIABLES)

self.session.run(tf.initialize\_variables(var\_lists))

#var\_lists = tf.get\_collection(tf.GraphKeys.GLOBAL\_VARIABLES)

#self.session.run(tf.variables\_initializer(var\_lists))

# make sure all variables are initialized

self.session.run(tf.assert\_variables\_initialized())

self.summary\_every = summary\_every

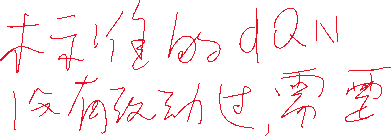
if self.summary\_writer is not None:

# graph was not available when journalist was created

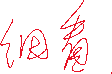
self.summary\_writer.add\_graph(self.session.graph)

self.summary\_every = summary\_every

def create\_variables(self):



# compute action from a state: a\* = argmax\_a Q(s\_t,a)



with tf.name\_scope("predict\_actions"):

# raw state representation

self.states = tf.placeholder(tf.float32, (None, self.state\_dim), name="states")

# initialize Q network

with tf.variable\_scope("q\_network"):

self.q\_outputs = self.q\_network(self.states)

# predict actions from Q network

self.action\_scores = tf.identity(self.q\_outputs, name="action\_scores")

tf.summary.histogram("action\_scores", self.action\_scores)

self.predicted\_actions = tf.argmax(self.action\_scores, dimension=1, name="predicted\_actions")

# estimate rewards using the next state: r(s\_t,a\_t) + argmax\_a Q(s\_{t+1}, a)

with tf.name\_scope("estimate\_future\_rewards"):

self.next\_states = tf.placeholder(tf.float32, (None, self.state\_dim), name="next\_states")

self.next\_state\_mask = tf.placeholder(tf.float32, (None,), name="next\_state\_masks")

if self.double\_q\_learning:

# reuse Q network for action selection

with tf.variable\_scope("q\_network", reuse=True):

self.q\_next\_outputs = self.q\_network(self.next\_states)

self.action\_selection = tf.argmax(tf.stop\_gradient(self.q\_next\_outputs), 1, name="action\_selection")

tf.summary.histogram("action\_selection", self.action\_selection)

self.action\_selection\_mask = tf.one\_hot(self.action\_selection, self.num\_actions, 1, 0)

# use target network for action evaluation

with tf.variable\_scope("target\_network"):

self.target\_outputs = self.q\_network(self.next\_states) \* tf.cast(self.action\_selection\_mask,

tf.float32)

self.action\_evaluation = tf.reduce\_sum(self.target\_outputs, axis=[1, ])

tf.summary.histogram("action\_evaluation", self.action\_evaluation)

self.target\_values = self.action\_evaluation \* self.next\_state\_mask

else:

# initialize target network

with tf.variable\_scope("target\_network"):

self.target\_outputs = self.q\_network(self.next\_states)

# compute future rewards

self.next\_action\_scores = tf.stop\_gradient(self.target\_outputs)

#self.target\_values = tf.reduce\_max(self.next\_action\_scores, axis=[1, ]) \* self.next\_state\_mask

self.target\_values = tf.reduce\_max(self.next\_action\_scores,

reduction\_indices=[1, ]) \* self.next\_state\_mask

tf.summary.histogram("next\_action\_scores", self.next\_action\_scores)

self.rewards = tf.placeholder(tf.float32, (None,), name="rewards")

self.future\_rewards = self.rewards + self.discount\_factor \* self.target\_values

# compute loss and gradients

with tf.name\_scope("compute\_temporal\_differences"):

# compute temporal difference loss

self.action\_mask = tf.placeholder(tf.float32, (None, self.num\_actions), name="action\_mask")

#self.masked\_action\_scores = tf.reduce\_sum(self.action\_scores \* self.action\_mask, axis=[1, ])

self.masked\_action\_scores = tf.reduce\_sum(self.action\_scores \* self.action\_mask, reduction\_indices=[1, ])

self.temp\_diff = self.masked\_action\_scores - self.future\_rewards

self.norm\_diff = tf.square(tf.sigmoid(self.masked\_action\_scores / 100.0) - tf.sigmoid(self.future\_rewards / 100.0))

#self.norm\_diff = tf.nn.sigmoid(tf.square(self.temp\_diff)/40000.0)

self.td\_loss = tf.reduce\_mean(self.norm\_diff) \* 20000.0

# regularization loss

q\_network\_variables = tf.get\_collection(tf.GraphKeys.TRAINABLE\_VARIABLES, scope="q\_network")

self.reg\_loss = self.reg\_param \* tf.reduce\_sum([tf.reduce\_sum(tf.square(x)) for x in q\_network\_variables])

# compute total loss and gradients

self.loss = self.td\_loss + self.reg\_loss

gradients = self.optimizer.compute\_gradients(self.loss)

# clip gradients by norm

for i, (grad, var) in enumerate(gradients):

if grad is not None:

gradients[i] = (tf.clip\_by\_norm(grad, self.max\_gradient), var)

# add histograms for gradients.

for grad, var in gradients:

tf.summary.histogram(var.name, var)

if grad is not None:

tf.summary.histogram(var.name + '/gradients', grad)

self.train\_op = self.optimizer.apply\_gradients(gradients)

# update target network with Q network

with tf.name\_scope("update\_target\_network"):

self.target\_network\_update = []

# slowly update target network parameters with Q network parameters

q\_network\_variables = tf.get\_collection(tf.GraphKeys.TRAINABLE\_VARIABLES, scope="q\_network")

target\_network\_variables = tf.get\_collection(tf.GraphKeys.TRAINABLE\_VARIABLES, scope="target\_network")

for v\_source, v\_target in zip(q\_network\_variables, target\_network\_variables):

# this is equivalent to target = (1-alpha) \* target + alpha \* source

update\_op = v\_target.assign\_sub(self.target\_update\_rate \* (v\_target - v\_source))

self.target\_network\_update.append(update\_op)

self.target\_network\_update = tf.group(\*self.target\_network\_update)

# scalar summaries

tf.summary.scalar("td\_loss", self.td\_loss)

#tf.summary.scalar("reg\_loss", self.reg\_loss)

tf.summary.scalar("total\_loss", self.loss)

tf.summary.scalar("exploration", self.exploration)

self.summarize = tf.summary.merge\_all()

self.no\_op = tf.no\_op()

def storeExperience(self, state, action, reward, next\_state, done):

# always store end states

if self.store\_experience\_cnt % self.store\_replay\_every == 0 or done:

self.replay\_buffer.add(state, action, reward, next\_state, done)

self.store\_experience\_cnt += 1

def eGreedyAction(self, states, explore=True):

if explore and self.exploration > random.random():

return random.randint(0, self.num\_actions - 1)

else:

return self.session.run(self.predicted\_actions, {self.states: states})[0]

def annealExploration(self, stategy='linear'):

ratio = max((self.anneal\_steps - self.train\_iteration) / float(self.anneal\_steps), 0)

self.exploration = (self.init\_exp - self.final\_exp) \* ratio + self.final\_exp

def updateModel(self, episode = -1):

# not enough experiences yet

print("compare ", self.replay\_buffer.count(), self.batch\_size)

if self.replay\_buffer.count() < self.batch\_size:

return

batch = self.replay\_buffer.getBatch(self.batch\_size)

states = np.zeros((self.batch\_size, self.state\_dim))

rewards = np.zeros((self.batch\_size,))

action\_mask = np.zeros((self.batch\_size, self.num\_actions))

next\_states = np.zeros((self.batch\_size, self.state\_dim))

next\_state\_mask = np.zeros((self.batch\_size,))

for k, (s0, a, r, s1, done) in enumerate(batch):

states[k] = s0

rewards[k] = r

action\_mask[k][a] = 1

# check terminal state

if not done:

next\_states[k] = s1

next\_state\_mask[k] = 1

# whether to calculate summaries

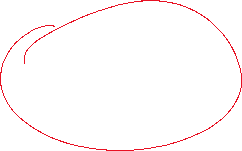
calculate\_summaries = self.train\_iteration % self.summary\_every == 0 and self.summary\_writer is not None

# perform one update of training

#direct\_r, nxt\_r, label\_r, now\_net\_r, diff, norm\_diff, cost, td\_cost, reg\_cost, \_, summary\_str = self.session.run([



cost, td\_cost, reg\_cost, \_, summary\_str = self.session.run([



#self.rewards,



#self.target\_values \* self.discount\_factor,

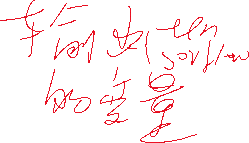
#self.future\_rewards,

#self.masked\_action\_scores,

#self.temp\_diff,

#self.norm\_diff,

self.loss,



self.td\_loss,

self.reg\_loss,

self.train\_op,

self.summarize if calculate\_summaries else self.no\_op

], {

self.states: states,

self.next\_states: next\_states,

self.next\_state\_mask: next\_state\_mask,

self.action\_mask: action\_mask,

self.rewards: rewards

})

'''

rewards\_out = open(rewards\_out\_path, 'a+')

if self.train\_iteration % 100 == 0:

for i in range(len(direct\_r)):

print("episode: ", episode, "iter: ", self.train\_iteration, "mini batch --- ", i, "direct\_r ",

direct\_r[i],

"nxt\_r: ", nxt\_r[i], "label\_r: ", label\_r[i], "now\_net\_r: ", now\_net\_r[i],

"tmpdiff: ", diff[i],

"norm\_diff", norm\_diff[i],



#"loss", cost[i],



#"state: ", states[i],



file=rewards\_out)

sys.stdout.flush()

rewards\_out.close()

'''

#if self.train\_iteration % 500:

# print('0000 : ', diff, file=logf)

# print('llll : ', norm\_diff, file=logf)

loss\_out = open(loss\_out\_path, "a+")

print("episode: ", episode, "iter: ", self.train\_iteration, "hjk loss is ----- ", cost, "hjk td\_loss is ----- ", td\_cost, "hjk reg\_loss is ----- ", reg\_cost, file=loss\_out)

sys.stdout.flush()

loss\_out.close()

# update target network using Q-network

self.session.run(self.target\_network\_update)

'''

# emit summaries

if calculate\_summaries:

self.summary\_writer.add\_summary(summary\_str, self.train\_iteration)

'''

self.annealExploration()

self.train\_iteration += 1

del batch, states, rewards, action\_mask, next\_states, next\_state\_mask

#del direct\_r, nxt\_r, label\_r, now\_net\_r, diff, norm\_diff

gc.collect()

#objgraph.show\_most\_common\_types(limit=50)

def save\_net(self, path):

saver = tf.train.Saver()

save\_path = saver.save(self.session, path)

print("Save to path: " + save\_path)