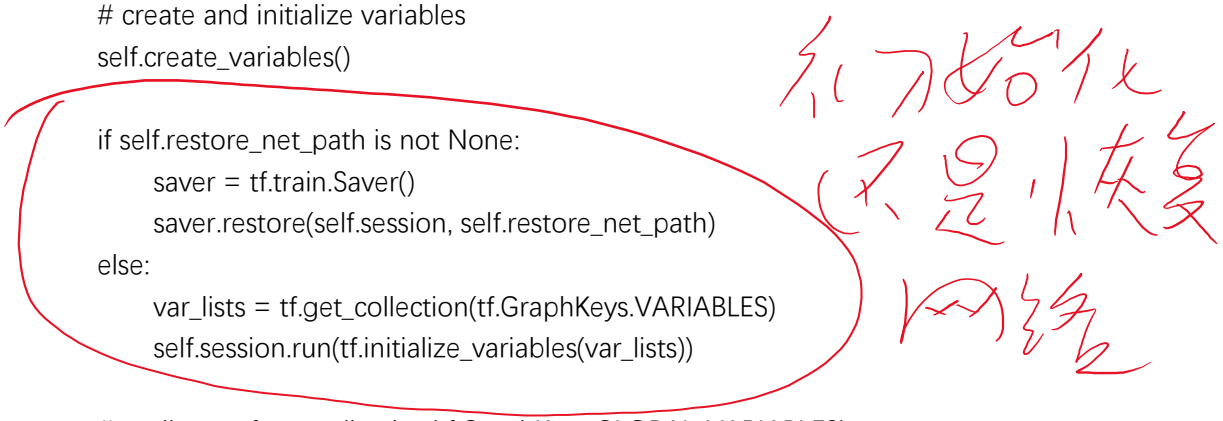
这个文件几乎没有改动

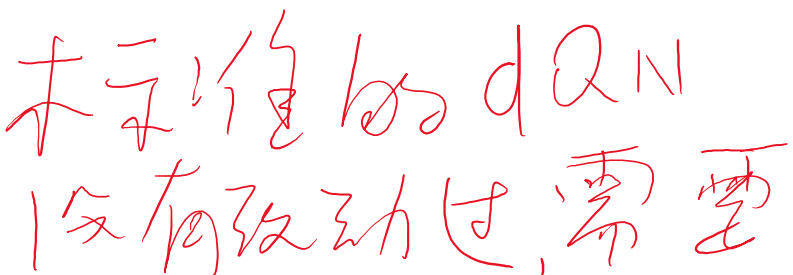
```
from __future__ import print_function
import random
import numpy as np
import tensorflow as tf
import gc
import sys
from replay_buffer import ReplayBuffer
from time import gmtime, strftime
logf = open("my_net22/log_" + strftime("%Y-%m-%d-%H-%M-%S", gmtime()) + ".txt", 'a+')
loss_out_path = "my_net22/loss_" + 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)
```

用于恢复网络

tensorflow的tensorboard相关，又用不着

```
        # 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):
```





又寸看论文仔细看。

```
        # 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)
```