```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
hindsight experience replay enhanced dqn agent for the 2dnav environment
"""

import numpy  as np
import torch.optim as optim
from   torch  import nn, load, FloatTensor, save, device, cuda, manual_seed
import sys
import copy
import random
#import pandas as pd
from   utils  import ValueNetwork

class Agent():
    def __init__(self, inputsize, mode="Train", seed=0, ep_decay_val=0.999, save_loc="",
                 max_ep=1000, memory=2**16, osize=4):
        manual_seed(seed)
        np.random.seed(seed)
        random.seed(seed)
        self.seed                       = seed
        self.inputsize                  = inputsize
        self.outputsize                 = osize
        self.learning_rate              = 1e-2
        self.mode                       = mode
        if mode == "Train":
            self.exploration            = 1.0
            self.explore_decay          = ep_decay_val
            self.load_model_flag        = False
        else:
            self.exploration            = 0.0
            self.explore_decay          = 0.0
            self.load_model_flag        = True
        self.gamma                      = 0.99
        self.memory_max_size            = memory
        self.batchsize                  = 2**12
        self.mem_states                 = []
        self.mem_target                 = []
        self.mem_action                 = []
        self.train_counter              = 0
        self.episodes                   = -1
        self.save_location              = save_loc
        self.max_episodes               = max_ep
        self.spawn_by                   = 0
        self.pol                        = 0
        self.cdown                      = 0
        '''
        HER-specific stuff
        '''
        self.goal_buf                   = [[0.0 for i in range(inputsize)]]
        self.max_goals                  =  8
        self.term_rew                   =  1
        self.goal_rew                   = [-np.inf]

        is_cuda = cuda.is_available()
        if is_cuda:
            #print("using cuda")
            self.device = device("cuda")
        else:
            #print("using cpu")
            self.device = device("cpu")

        self.qnet                       =
ValueNetwork(int(2*self.inputsize),self.outputsize).to(self.device)
        self.criterion                  = nn.MSELoss()
```

```
        self.optimizer                      = optim.Adam(self.qnet.parameters(),
lr=self.learning_rate)
        #self.optimizer                      = optim.SGD(self.qnet.parameters(),
lr=self.learning_rate, momentum=0.9)
        print("[HER/" + str(self.seed) + "]: Model created")
        #print(self.qnet)
        if self.load_model_flag == True:
            modelpath = save_loc + "/qnet.pt"
            print("Looking for qnet.pt in " + modelpath)
            self.load_weights(modelpath)
            print("Success")
            self.exploration   = 0.0
            self.explore_decay = 0.0
            print("Deactivated exploration noise, since loaded a trained model")
        #print("")

    def reset(self):
        self.ep_states = []
        self.ep_action = []
        self.ep_reward = []
        self.ep_target = []
        self.loss      = np.inf
        self.episodes += 1
        self.tot_rew   = 0
        self.goal      = self.sample_goals()

    def sample_goals(self):
        idx            = np.random.randint(0,len(self.goal_buf))
        return self.goal_buf[idx]  Which goal to aim for

    def load_weights(self, wt_path):
        print("Using Saved Weights")
        self.qnet.load_state_dict(load(wt_path))
        self.qnet.eval()

    def choose_action(self, state, reward, t):
        state.extend(copy.deepcopy(self.goal))
        self.ep_states.append(copy.deepcopy(state))
        if t > 0: self.ep_reward.append(reward)
        state  = np.array(state).reshape(2*self.inputsize,)
        qval   =
self.qnet(FloatTensor(state).to(self.device)).detach().cpu().numpy().reshape(self.outputsize,)
        if np.random.random() < self.exploration:
            action = np.random.randint(self.outputsize)
        else:
            action = np.argmax(qval)
        self.ep_action.append(action)
        self.ep_target.append(copy.deepcopy(qval))
        return action

    def terminate(self, reward, ep):
        self.ep_reward.append(reward)
        self.tr_reward    = self.ep_reward
        self.exploration *= self.explore_decay
        rewards     = [reward] # Terminal value
        next_reward = reward
        self.tot_rew = sum(self.ep_reward)
        for r in range(1,len(self.ep_reward)):
            # Compute rewards backward
            next_reward = self.ep_reward[len(self.ep_reward)-1-r] + self.gamma*next_reward
            rewards.insert(0,next_reward)
        for t in range(len(self.ep_reward)):
            # Attach to targets
            self.ep_target[t][self.ep_action[t]] = rewards[t]
        if self.mode == "Train":
            self.mem_states.extend(copy.deepcopy(self.ep_states))
```

```
            self.mem_action.extend(copy.deepcopy(self.ep_action))
            self.mem_target.extend(copy.deepcopy(self.ep_target))
            self.extend_with_HER(reward)
            if not (len(self.mem_action)==len(self.mem_states)==len(self.mem_target)):
                sys.exit("Some error with memory buffer sizes")
            self.train()

    def extend_with_HER(self, reward):
        # Final state is goal
        goal = self.ep_states[-1][0:self.inputsize]
        #self.goal_buf.append(copy.deepcopy(goal))
        #sys.exit()

        if len(self.goal_buf) >= self.max_goals:
            if reward > min(self.goal_rew):
                id_replace  = np.argmin(np.array(self.goal_rew))
                self.goal_rew[id_replace] = reward
                self.goal_buf[id_replace] = copy.deepcopy(goal)
                #print("[HER/" + str(self.seed) + "]: EP=" + str(self.episodes+1) + ":
Goal is " + str(np.round(np.array(goal),decimals=3)))
                self.append_memory(goal, reward)
        else:
            self.goal_buf.append(copy.deepcopy(goal))
            self.goal_rew.append(reward)
            #print("[HER/" + str(self.seed) + "]: EP=" + str(self.episodes+1) + ": Goal is
" + str(np.round(np.array(goal),decimals=3)))
            self.append_memory(goal, reward)

    def append_memory(self, goal, reward):
        rewards     = [reward+self.term_rew] # Terminal value, including end state as goal
state
        next_reward =  reward+self.term_rew
        her_rewards = copy.deepcopy(self.ep_reward)
        her_states  = copy.deepcopy(self.ep_states)
        her_targets = copy.deepcopy(self.ep_target)
        for r in range(1,len(her_rewards)):
            # Compute rewards backward
            if not (False in [her_states[len(her_rewards)-1-r][i] == goal[i] for i in
range(self.inputsize)]):
                # Matching the goal state
                her_reward = self.term_rew
            else:
                her_reward = 0
            next_reward = her_reward + her_rewards[len(her_rewards)-1-r] +
self.gamma*next_reward
            rewards.insert(0,next_reward)
        self.ep_reward  = copy.deepcopy(rewards)
        for t in range(len(self.ep_reward)):
            # Create HER memory
            her_states [t][self.inputsize:]   = goal
            her_targets[t]                    =
self.qnet(FloatTensor(her_states[t]).to(self.device)).detach().cpu().numpy().reshape(self.outputsize,)
            her_targets[t][self.ep_action[t]] = rewards[t]
        self.mem_states.extend(copy.deepcopy(her_states))
        self.mem_action.extend(copy.deepcopy(self.ep_action))
        self.mem_target.extend(copy.deepcopy(her_targets))
        #to_write = np.concatenate((np.array(self.mem_states),
        #
np.array(self.mem_action).reshape(len(self.mem_action),1),
        #                           np.array(self.mem_target)),axis=1)
        #pd.DataFrame(to_write).to_csv("../OP/debug-"+str(self.seed)+".csv")

    def train(self):
        if len(self.mem_states) > self.memory_max_size:
            # Clear some memory if exceeded maximum buffer size
            overflow        = len(self.mem_states) - self.memory_max_size
```

```
            self.mem_states = self.mem_states[overflow:] # s
            self.mem_action = self.mem_action[overflow:]    # a
            self.mem_target = self.mem_target[overflow:]    # target q-values
        for i in range(2):
            self.optimizer.zero_grad()
            # forward + backward + optimize
            batch   = min(self.batchsize,len(self.mem_states))
            if batch > 0:
                self.train_counter += 1
                idx     = random.choices(range(len(self.mem_states)), k=batch)
                inputs  = FloatTensor(np.array(self.mem_states)[idx]).to(self.device)
                outputs = self.qnet(inputs)
                #targets = np.array(self.spl_target)[idx]
                targets = copy.deepcopy(outputs.detach().numpy())
                temptrg = np.array(self.mem_target)[idx]
                actions = [self.mem_action[idx[x]] for x in range(len(idx))]
                for x in range(len(idx)):
                    targets[x,actions[x]] = temptrg[x,actions[x]]
                loss    = self.criterion(FloatTensor(outputs).to(self.device),
FloatTensor(targets).to(self.device))

                self.loss = float(loss.item())
                loss.backward()
                self.optimizer.step()
                if self.train_counter%250 == 0 or self.episodes==self.max_episodes-1:
                    save(self.qnet.state_dict(), self.save_location + "/models/qnet-
her-"+str(self.seed)+".pt")
        if self.episodes%250 == 0 or self.episodes==self.max_episodes-1:
            print("[HYP/" + str(self.seed) + "]: EP=" + str(self.episodes+1) + ", mem = "
+ str(len(self.mem_states)))
```