```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
vanilla 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
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

        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(self.inputsize,self.outputsize).to(self.device)
        self.criterion                      = nn.MSELoss()
        self.optimizer                      = optim.Adam(self.qnet.parameters(),
lr=self.learning_rate)
        print("[VAN/" + 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

    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):
        self.ep_states.append(copy.deepcopy(state))
        if t > 0: self.ep_reward.append(reward)
        state = np.array(state).reshape(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
        #if ep <= self.max_episodes - 500:
        #    self.exploration = max(0.05, self.exploration)
        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))
            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 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-
van-"+str(self.seed)+".pt")
        if self.episodes%250 == 0 or self.episodes==self.max_episodes-1:
            print("[VAN/" + str(self.seed) + "]: EP=" + str(self.episodes+1) + ",mem = " +
str(len(self.mem_states)))
```