代码说明文档

Q-learning

由于q-learning算法和sarsa算法仅仅在下一步动作的选择策略上有所区别，而其他框架一致，我们这里主要框架仅以q-learning模块为例

执行顺序：

执行 train.py模块即可完成全部的过程；

修改参数：

- 训练轮数：直接在train.py的train函数中修改
- 地图尺寸及模式：地图模块进行修改
- 修改地图尺寸后，需要修改tabular_q_agent.py模块test模块中的taction模块，有注释，可以参考

模型初始化

utils.py：

```
import random
def set_global_seeds(i):
    try:
        import tensorflow as tf
    except ImportError:
        pass
    else:
        tf.set_random_seed(i)
    try:
        import numpy as np
    except ImportError:
        pass
    else:
        np.random.seed(i)
    random.seed(i)
```

init.py：

```
from .utils import set_global_seeds

set_global_seeds(0)
```

通过设置随机种子保证了在i值一定的时候随机生成的数据一致，保证了全局实验的可复现。

地图模块

map.py：

```
custom_map = [
'SFFHF',
'HFHFF',
'HFFFH',
'HHHFH',
'HFFFG'
]
custom_map=None
map_config={'desc':custom_map,'map_name':'8x8','is_slippery':False}
```

custom_map实现自定义地图的功能，map_config设置一些无参数传入状态下地图参数的默认值

算法模块

tabular_q_agent.py:

我们仅介绍算法的一些核心模块，保证能够最快的复现实验过程

导入包及功能：

```
from collections import defaultdict
# 一个有用的数据结构，dict的子类，用于创建默认值为零的字典

import functools
# 一个重要的功能是提供了 functools.wraps 装饰器，用于更新被装饰函数的元数据，以便更好地保留原函数的信息。

import pickle
# pickle导入导出

import numpy as np
import time

from gym.spaces import discrete
# gym 提供了 gym.spaces 模块来处理不同类型的状态空间和动作空间。
# gym.spaces 模块中的 discrete 类是用于定义离散型空间的一种，它表示一个有限的整数集合，代表了离散的状态或动作空间。
# 可以用数来表示离散空间的动作范围

class UnsupportedSpace(Exception):
    pass
```

接下来介绍算法类主要的四个模块

init:实现初始化功能

if not isinstance（A，B）检查A是否是B的类型，确保观察到的空间和行为空间都是离散的空间

```
    if not isinstance(observation_space, discrete.Discrete):
        raise UnsupportedSpace('Observation space {} incompatible with {}. (Only
supports Discrete observation spaces.)'.format(observation_space, self))
```

存储一些重要的信息，初始Q表的平均值，方差，学习率，探索率，折扣率，最大探索步数

```
    self.config = {
        "init_mean": 0.0,      # Initialize Q values with this mean
        "init_std": 0.0,       # Initialize Q values with this standard deviation
        "learning_rate": 0.5,
        "eps": 0.05,            # Epsilon in epsilon greedy policies
        "discount": 0.99,
        "n_iter": 10000}        # Number of iterations
```

创建generate_zeros的部分应用版本，将其中的参数n值具体为某个数

```
    self.q = defaultdict(functools.partial(generate_zeros, n=self.action_n))
```

act: 采取行为时有概率进行探索，保证了算法不至于陷入鼠目寸光陷阱

eps: 探索率，探索失败时采取贪心策略

observation: 当前的观测空间

```
    # epsilon greedy.
    action = np.argmax(self.q[observation]) if np.random.random() > eps else
self.action_space.sample()
```

learn:进行多轮训练更新Q表的过程

obs:当前状态 obs2：下一步状态（位置）

设置回报 # Get negative reward every step if reward == 0: reward = -0.005

```
        # if agent sucked at same position, punish it
        if obs == obs2:
            reward = -0.01

        # if agent fill to hole then die, punish it
        if done and not reward:
            reward = -1
```

```
        future = 0.0
        if not done:
            future = np.max(self.q[obs2])
        # 没有到下一步就会采取贪心策略
```

Q-learning算法的更新策略

```
        self.q[obs][action] = (1 - learning_rate) * self.q[obs][action] +
learning_rate * (reward + self.config["discount"] * future)

        rAll += reward
        step_count += 1
        # 记录最终的回报和步数
```

test模块：利用我们的Q表控制飞机飞跃冰湖的过程

```
    # 这里为了更好的保证飞机的运行，不至于驶出地图外，我们增加了一些限制，
    # 但事实上，由于算法正确，完全可以去掉这一部分限制
    maplocationX=0
    maplocationY=0
```

飞机的起飞动作应该在后续命令动作之前

```
    self.st.send().takeoff(50)
    time.sleep(3)
```

贪心策略确定下一步命令

```
        action = self.act(obs, eps=0)
        obs2, reward, done, _ = env.step(action)
```

这里为了保证在stochastic模式下也能够正常运行，我们通过人物实际的位置来确定实际执行命令，并且依此给飞机发送命令

```
        dis=obs2-pos
        pos=obs2
        # dis后面的限制值应该根据地图大小进行修改，如果为4x4则改成dis==-4，下同
        if dis==-8:
            taction=3
        elif dis==8:
```

```
            taction=1
        elif dis==-1:
            taction=0
        elif dis==1:
            taction=2
        else:
            taction=-1
```

给飞机输送命令

```
            if taction == 3:  # up
                if maplocationY > 0:
                    self.st.send().forward(50)
                    time.sleep(3)
                    print("tag_id==",end="")
                    print(self.st.vision_sensor_info().tag_id)
                    time.sleep(1)
                    maplocationY-=1
                else:
                    maplocationY=0
                    pass
```

到达降落

```
        if done:
            self.st.send().land()
            time.sleep(3)
            # break
            if not reward:
                return 0,t+1
            return 1,t+1
```

train模块：执行指令

导入包：

```
from tqdm import tqdm
# tqdm模块的作用是将我们的进程进行实时的展示

from terminaltables import AsciiTable
# 这一个主要用来美化我们的输出结果，创建各种表格，完成结果输出的美化

from .tabular_q_agent import TabularQAgent
# 最最主要的算法模块
```

```
from .map import map_config
# 记录了我们需要使用的地图的信息
```

为了保证Q-learning算法更加有效，对学习率等参数做衰减操作

```
def exponential_decay(starter_learning_rate, global_step, decay_step,
                  decay_rate, mini_value=0.0):
    decayed_learning_rate = starter_learning_rate *
math.pow(decay_rate,math.floor(global_step / decay_step))

    return decayed_learning_rate if decayed_learning_rate > mini_value else
mini_value

    # 这里是后面训练过程中的操作
    learning_rate = exponential_decay(0.9, episode, 1000, 0.99)
    eps_rate = exponential_decay(1.0, episode, 1000, 0.97, 0.001)
```

保存我们的参数，保证结果可复现：

```
def save_rewards_as_pickle(rewards, filename='q_learning-rewards.pkl'):
with open(filename, 'wb') as file:
    pickle.dump(rewards, file)
```

结果展示的美化：

```
table_header = ['Episode', 'learning_rate', 'eps_rate', 'reward', 'step']
rewards = []
table_data = [table_header]

table_data.append([episode, round(learning_rate,3), round(eps_rate,3),
round(all_reward,3), step_count])
table = AsciiTable(table_data)
# 这里语句的作用就是直观显示每一轮训练过程的数据
tqdm.write(table.table)
# 实时化训练过程
```

通过和归一化的矩阵做卷积函数实现回报的归一化，并且将结果展示出来，也就是我们看到的那张收敛图

```
smoothed_data = np.convolve(rewards, np.ones(window_size)/window_size,
mode='valid')
```

其次就是调用具体的算法函数来实现过程，不再赘述

Sarsa算法

tabular_q_agent.py的learn函数中：

```
future = self.q[obs2][action2]
# 和Q-learning算法就只有这一步区别，下一步动作的选取上的区别
```

DQN算法

执行过程

- env：设置地图
- dqn.run(times):times表示训练轮数

导入包：

```
import tensorflow.compat.v1 as tf
# 现有tensorflow版本都是2.x版本，这里提供与 TensorFlow 1.x 版本兼容的接口
import numpy as np
import gym
import matplotlib.pyplot as plt
```

具体网络DQN的搭建：

init初始化：

```
self.nstate=nstate   # 状态空间的维度
self.naction=naction  # 动作空间的维度
self.sess = tf.Session()  # tensorflow会话，用于执行计算图
self.memcnt=0      # 记录回放缓冲区中当然存储的数据数量
self.BATCH_SIZE = 64  # 每次从缓冲区中采样量的大小
self.LR = 0.0012                    # learning rate
self.EPSILON = 0.92                 # greedy policy
self.GAMMA = 0.9999                 # reward discount
self.MEM_CAP = 2000     # 回放缓冲区大小
self.mem= np.zeros((self.MEM_CAP, self.nstate * 2 + 2))     # initialize memory
self.updataT=150       # 更新目标网络的频率
self.built_net()        # 方法用于构建神经网络
```

搭建网络： 评估网络和目标网络

定义状态、动作、奖励、下一状态的占位符，用于输入各个数据

```
self.s = tf.placeholder(tf.float64, [None,self.nstate])
self.a = tf.placeholder(tf.int32, [None,])
self.r = tf.placeholder(tf.float64, [None,])
self.s_ = tf.placeholder(tf.float64, [None,self.nstate])
```

定义两个网络

训练网络：

l_eval: 代表评估网络的隐藏层，具有10个神经元，激活函数为 ReLU。

self.q: 代表 Q-value 的输出层，其神经元数量为 self.naction，即动作的数量。这是代理根据当前状态估计的 Q-value。

```
with tf.variable_scope('q'):                                    # evaluation network
l_eval = tf.layers.dense(self.s, 10, tf.nn.relu,
kernel_initializer=tf.random_normal_initializer(0, 0.1))
self.q = tf.layers.dense(l_eval, self.naction,
kernel_initializer=tf.random_normal_initializer(0, 0.1))
```

目标网络：

l_target: 代表目标网络的隐藏层，具有10个神经元，激活函数为 ReLU。

q_next: 代表目标网络的输出层，其神经元数量为 self.naction。目标网络的参数在训练过程中不会更新（trainable=False）。

```
 with tf.variable_scope('q_next'):                # target network, not to train
    l_target = tf.layers.dense(self.s_, 10, tf.nn.relu, trainable=False)
    q_next = tf.layers.dense(l_target, self.naction, trainable=False)
```

计算目标的q-value和当前估计的q-value

```
    q_target = self.r + self.GAMMA * tf.reduce_max(q_next, axis=1)    #q_next:
shape=(None, naction),
    # 计算当前估计的q-value
    a_index=tf.stack([tf.range(self.BATCH_SIZE,dtype=tf.int32),self.a],axis=1)
    q_eval=tf.gather_nd(params=self.q,indices=a_index)
```

定义损失、优化器、初始化

```
loss=tf.losses.mean_squared_error(q_target,q_eval)
# 损失函数
```

```
self.train=tf.train.AdamOptimizer(self.LR).minimize(loss)
#  q现实target_net- Q估计，定义优化操作
self.sess.run(tf.global_variables_initializer())
# 初始化变量，保证模型可以正确运行
```

greedy策略选择动作

```
if  np.random.uniform(0.0,1.0)<self.EPSILON:
    action=np.argmax( self.sess.run(self.q,feed_dict={self.s:fs}))
else:
    action=np.random.randint(0,self.naction)
```

learn

每次训练前，先判断是否需要对目标网络进行更新，判断完成后再从回放缓冲区随机选择一批样本，利用这批样本对评估网络进行一次优化操作

```
def learn(self):
    if(self.memcnt%self.updataT==0):
        t_params = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES,
scope='q_next')
        e_params = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope='q')
        self.sess.run([tf.assign(t, e) for t, e in zip(t_params, e_params)])
    rand_indexs=np.random.choice(self.MEM_CAP,self.BATCH_SIZE,replace=False)
    temp=self.mem[rand_indexs]
    bs = temp[:,0:self.nstate]#.reshape(self.BATCH_SIZE,NSTATUS)
    ba = temp[:,self.nstate]
    br = temp[:,self.nstate+1]
    bs_ = temp[:,self.nstate+2:]#.reshape(self.BATCH_SIZE,NSTATUS)
    self.sess.run(self.train, feed_dict=
{self.s:bs,self.a:ba,self.r:br,self.s_:bs_})
```

存储数据进入回放缓冲区

```
def storeExp(self,s,a,r,s_):
fs = np.zeros(self.nstate)
fs[s] = 1.0                         # ONE HOT
fs_ = np.zeros(self.nstate)
fs_[s_] = 1.0                             # ONE HOT
self.mem[self.memcnt%self.MEM_CAP]=np.hstack([fs,a,r,fs_])
self.memcnt+=1
# one-hot编码后，存储到回放缓冲区，采用循环队列的模式，确保不会超过回放缓冲区的容量
```

展示结果

```
def show(self):
    print("show")
    obs = env.reset()
    env.render('human')
    for t in range(10000):
        env.render('human')
        action = dqn.choose_action(obs)
        obs2, reward, done, _ = env.step(action)
        env.render('human')
        if done:
            break
        obs = obs2
```

集成所有模块运行

记录用数据

```
    cnt_win = 0 # 记录最近50次中完成任务的次数
    winrate_recorder = 0 # 记录最近10回合中成功完成任务的次数
    all_r=0.0  # 记录所有回合的累计奖励
    win_rate=[]
```

每次训练后存储

```
a=self.choose_action(s)
s_,r,done,_=env.step(a)
all_r+=r
self.storeExp(s,a,r,s_)
```

经验池满则网络进行一次学习：

```
 if(self.memcnt>self.MEM_CAP):
    self.learn()  # 经验池满，则进行一次学习
```

更新记录用数据

```
if(done):
    if(s_==self.nstate-1):
        cnt_win+=1.0
        winrate_recorder+=1.0
```

根据最近50次的任务完成率对贪心策略的执行率做出调整，越大越不可能执行随机动作

```
if (cnt_win / 50 > 0.4):
        self.EPSILON += 0.01
elif (cnt_win / 50 > 0.2):
        self.EPSILON += 0.005
elif (cnt_win / 50 > 0.1):
        self.EPSILON += 0.003
elif (cnt_win / 50 > 0.05):
        self.EPSILON += 0.001
```

绘制结果曲线

```
x_axis = [i * 10 for i in range(len(win_rate))]
plt.plot(x_axis, win_rate)
plt.show()
# 绘制训练过程中平均胜率的曲线
```