

**中国石油大学（北京）克拉玛依校区**

统计机器学习

实 验 报 告

**课程名称：** 统计机器学习

**班 级：**  数据18-2班

**学生姓名：** 黄克伦

**学 号：**  2018015331

垃圾邮件检测

一、实验时间和地点

C5Ⅱ区320、 6月2日

二、实验环境

硬件：处理器：AMD R5 3550H 显卡：Vega 8

环境： 操作系统：win10 pro 19041.985 IDE：Visual Studio Code

语言：Python3.7.0 64bit

支持库：numpy1.20.3

数据集：spam中25个垃圾邮件、ham中25个正常邮件。

三、算法介绍

P(c|w)=。总共就分为2类：abusive和非abusive。C1代表abusive即spam。每次分类只需比较P(w|c0)和P(w|c1)的大小，谁大就归在哪一类，因此在不等式中去掉相同的分母，每次只需比较P(w|c1)P(c1)和P(w|c0)P(c0)的大小。由于采用贝叶斯分类简化算法，P(w|c0)拆开成P(w1|c0)\* P(w2|c0)…\* P(wn|c0)。为了使分类器能更好地利用特征，P(w0|c)的幂是单词在vocabulary List中出现的次数。为了防止上式中出现0和小数连乘造成下溢出，分子从1开始计数坟分母从2开始计数，使用lg(P)而不用P。因此每次计算所属概率时，比较的值是lg(P(w1|c0))+ lg(P(w2|c0))+…+ lg(P(wn|c0))+ lg(P(c0))。

四、程序流程

textParse()将每封邮件拆分成多个words。小于2个letter的扔掉。所有words是小写。下滑线保留了，网站链接分成多个words。

createVocabList()将所有邮件拆分后的words都放在一个集合中（不重复）。

trainNB0()根据所有的邮件分别得到P(w|c1)的list、P(w|c0)的list和P(c1)。每个邮件被拆分成words后，根据vocabulary list得到这个邮件的特征向量，这里用的是词袋模型。在每个分类下加和所有的词袋模型的特征向量，最后利用numpy特性，将这个list同时除以words总数就得到了后验概率的list。

classifyNB()分类。哪个值大就归哪一类。

spamTest()开始读取spam和ham中的邮件，每个邮件拆分成词，变成特征向量，最终所有邮件组合成由特征值向量组成的矩阵。Vocabulary list是由所有words组成的。留存交叉验证：随机选40封用来计算P(w|c0)的list，剩下10封用来测试。由于是监督学习，每次测试只要比较分类器的输出值与标签是否一致，标签存储在trainClass的list中。最后计算正确率。

五、仿真实验结果及分析



错误率最高时为0.1，可以看到效果非常好。考虑到实际生活中，将垃圾邮件误判为正常邮件要比将正常邮件归到垃圾邮件好。因此，在classifyNB()函数中直接比较P(c1)和P(c0)的大小来判断不太好，可以设计成P(c1)/P(c0)-1大于某个阈值时才归为垃圾邮件。这样可能会增加错误率，但至少比将正常邮件归为垃圾邮件而耽误事情好。

六、代码

# https://github.com/pbharrin/machinelearninginaction/tree/master/Ch04

# 不能直接用ham中的txt，第23个文件要修改一下（去掉问号），不然会报错

# spamTest中的trainingSet要改为list（range（50））

# textParse函数中的\W\*去掉\*

import numpy as np

import math

def textParse(bigString):  # input is big string, #output is word list

    import re

    listOfTokens = re.split(r'\W', bigString)

    return [tok.lower() for tok in listOfTokens if len(tok) > 2]

def createVocabList(dataSet):

    vocabSet = set([])  # create empty set

    for document in dataSet:

        # union of the two sets, will filter the same element

        vocabSet = vocabSet | set(document)

    return list(vocabSet)

def bagOfWords2VecMN(vocabList, inputSet):

    returnVec = [0]\*len(vocabList)

    for word in inputSet:

        if word in vocabList:

            returnVec[vocabList.index(word)] += 1

    return returnVec

def trainNB0(trainMatrix, trainCategory):

    numTrainDocs = len(trainMatrix)

    numWords = len(trainMatrix[0])

    pAbusive = sum(trainCategory)/float(numTrainDocs)  # 计算 P(c1)

    p0Num = np.ones(numWords)

    p1Num = np.ones(numWords)

    p0Denom = 2.0

    p1Denom = 2.0  # change to 2.0

    for i in range(numTrainDocs):

        if trainCategory[i] == 1:

            p1Num += trainMatrix[i]

            p1Denom += sum(trainMatrix[i])

        else:

            p0Num += trainMatrix[i]

            p0Denom += sum(trainMatrix[i])

    p1Vect = np.log(p1Num/p1Denom)  # change to log() in case of overflow

    p0Vect = np.log(p0Num/p0Denom)  # change to log() in case of overflow

    return p0Vect, p1Vect, pAbusive

def classifyNB(vec2Classify, p0Vec, p1Vec, pClass1):

    p1 = sum(vec2Classify \* p1Vec) + np.log(pClass1)  # element-wise mult

    p0 = sum(vec2Classify \* p0Vec) + np.log(1.0 - pClass1)

    if p1 > p0:

        return 1

    else:

        return 0

def spamTest():

    docList = []

    classList = []

    fullText = []

    for i in range(1, 26):

        wordList = textParse(open('./email\\spam\\%d.txt' % i).read())

        docList.append(wordList)

        fullText.extend(wordList)

        classList.append(1)

        wordList = textParse(open('./email\\ham\\%d.txt' % i).read())

        docList.append(wordList)

        fullText.extend(wordList)

        classList.append(0)

    vocabList = createVocabList(docList)  # create vocabulary

    trainingSet = list(range(50))

    testSet = []  # create test set

    for i in range(10):

        randIndex = int(np.random.uniform(0, len(trainingSet)))

        testSet.append(trainingSet[randIndex])

        del(trainingSet[randIndex])

    trainMat = []

    trainClasses = []

    for docIndex in trainingSet:  # train the classifier (get probs) trainNB0

        trainMat.append(bagOfWords2VecMN(vocabList, docList[docIndex]))

        trainClasses.append(classList[docIndex])

    p0V, p1V, pSpam = trainNB0(trainMat, trainClasses)

    errorCount = 0

    for docIndex in testSet:  # classify the remaining items

        wordVector = bagOfWords2VecMN(vocabList, docList[docIndex])

        if classifyNB(wordVector, p0V, p1V, pSpam) != classList[docIndex]:

            errorCount += 1

            print("classification error", docList[docIndex])

            print('Speculation is: ', classifyNB(wordVector, p0V, p1V, pSpam))

            print('the truth is: ', classList[docIndex])

    print('the error rate is: ', float(errorCount)/len(testSet))

spamTest()

自动爬山小车

一、实验时间和地点

C5Ⅱ区320、 6月9日

二、实验环境

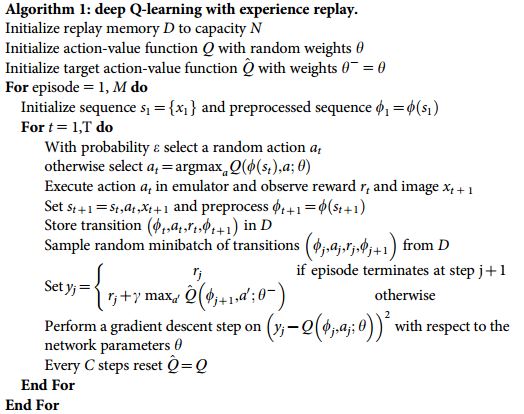
硬件：处理器：AMD R5 3550H 显卡：Vega 8

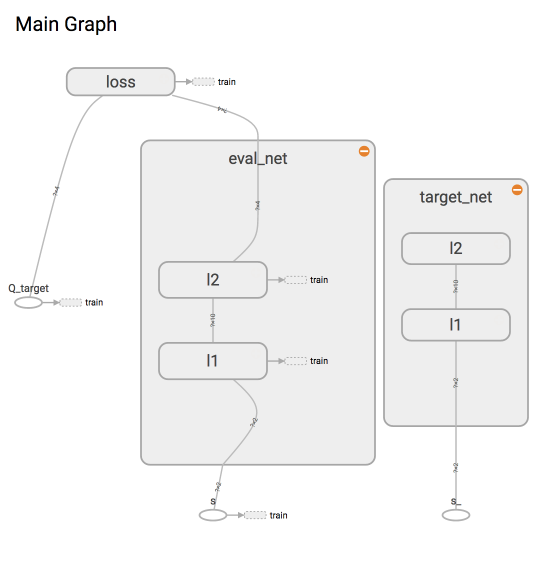
环境： 操作系统：win10 pro 19041.985 IDE：Visual Studio Code

语言：Python3.7.0 64bit

支持库：numpy1.20.3、tensorflow1.14.0、gym0.18.3、pandas1.1.5

三、算法介绍

Q学习、off-policy、e-贪心策略：小于e时执行探索策略，大于e时执行利用策略，

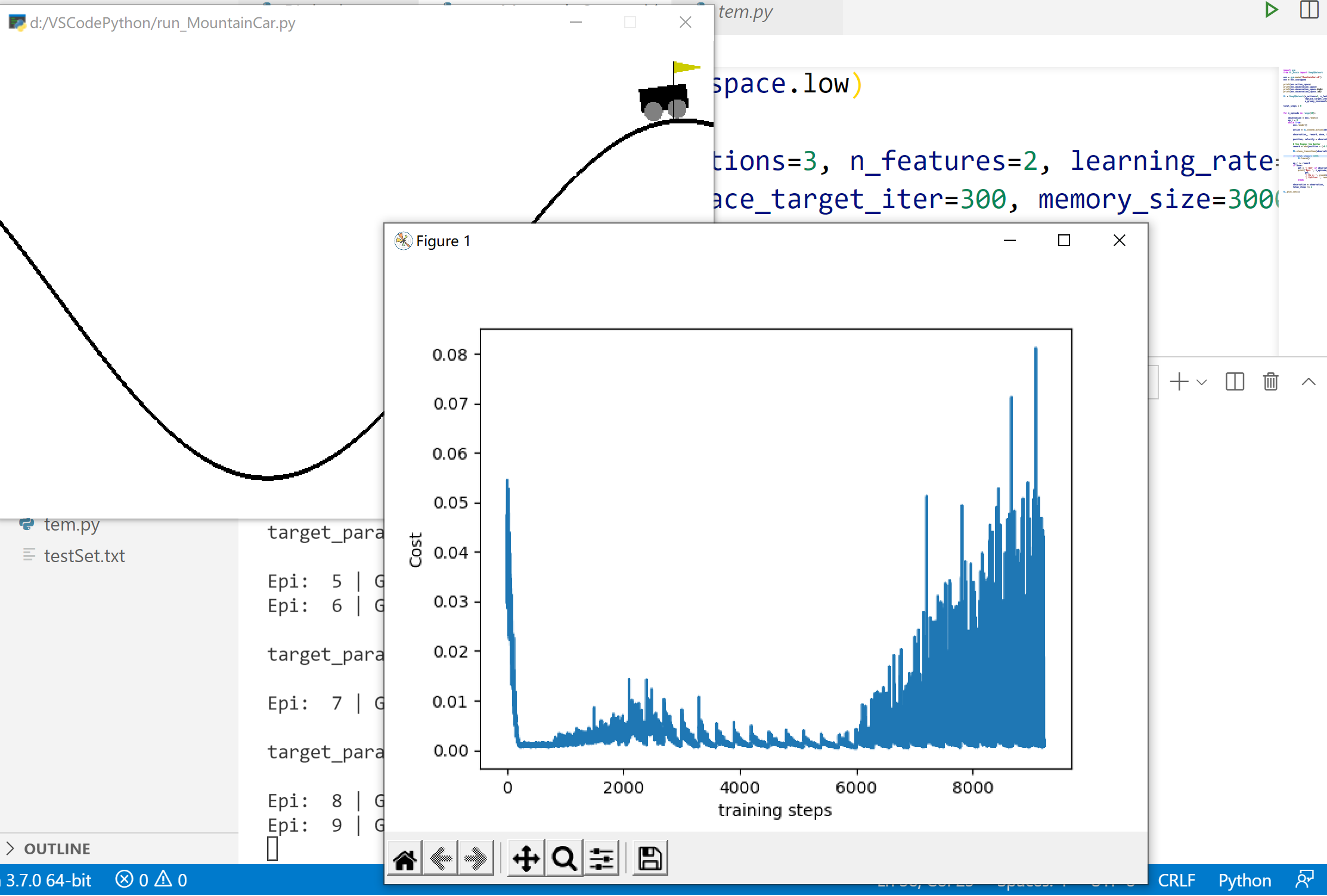


target\_net 用于预测 q\_target 值, 他不会及时更新参数. eval\_net 用于预测 q\_eval, 这个神经网络拥有最新的神经网络参数. 不过这两个神经网络结构是完全一样的, 只是里面的参数不一样。target\_net 是 eval\_net 的一个历史版本, 拥有 eval\_net 很久之前的一组参数, 而且这组参数被固定一段时间, 然后再被 eval\_net 的新参数所替换。 而 eval\_net 是不断在被提升的,所以是一个可以被训练的网络 trainable=True. 而 target\_net 的 trainable=False.

四、程序流程

每当target net训练了很多个batch之后，就会直接把eval net的所有参数照搬到target net,上来。最后，抽取缓冲区的样本进行学习训练，更新网络参数。

五、仿真实验结果及分析



可以看出曲线并不是平滑下降的, 这是因为 DQN 中的 input 数据是一步步改变的, 而且会根据学习情况, 获取到不同的数据。

六、代码

RL\_brain.py

# https://github.com/MorvanZhou/Reinforcement-learning-with-tensorflow/tree/master/contents/6\_OpenAI\_gym

import numpy as np

import pandas as pd

import tensorflow as tf

# Deep Q Network off-policy

class DeepQNetwork:

    def \_\_init\_\_(

            self,

            n\_actions,

            n\_features,

            learning\_rate=0.01,

            reward\_decay=0.9,

            e\_greedy=0.9,

            replace\_target\_iter=300,

            memory\_size=500,

            batch\_size=32,

            e\_greedy\_increment=None,

            output\_graph=False,

    ):

        self.n\_actions = n\_actions

        self.n\_features = n\_features

        self.lr = learning\_rate

        self.gamma = reward\_decay

        self.epsilon\_max = e\_greedy

        self.replace\_target\_iter = replace\_target\_iter

        self.memory\_size = memory\_size

        self.batch\_size = batch\_size

        self.epsilon\_increment = e\_greedy\_increment

        self.epsilon = 0 if e\_greedy\_increment is not None else self.epsilon\_max

        # total learning step

        self.learn\_step\_counter = 0

        # initialize zero memory [s, a, r, s\_]

        self.memory = np.zeros((self.memory\_size, n\_features \* 2 + 2))

        # consist of [target\_net, evaluate\_net]

        self.\_build\_net()

        t\_params = tf.get\_collection('target\_net\_params')

        e\_params = tf.get\_collection('eval\_net\_params')

        self.replace\_target\_op = [tf.assign(t, e)

                                  for t, e in zip(t\_params, e\_params)]

        self.sess = tf.Session()

        if output\_graph:

            # $ tensorboard --logdir=logs

            # tf.train.SummaryWriter soon be deprecated, use following

            tf.summary.FileWriter("logs/", self.sess.graph)

        self.sess.run(tf.global\_variables\_initializer())

        self.cost\_his = []

    def \_build\_net(self):

        # ------------------ build evaluate\_net ------------------

        self.s = tf.placeholder(

            tf.float32, [None, self.n\_features], name='s')  # input

        self.q\_target = tf.placeholder(

            tf.float32, [None, self.n\_actions], name='Q\_target')  # for calculating loss

        with tf.variable\_scope('eval\_net'):

            # c\_names(collections\_names) are the collections to store variables

            c\_names, n\_l1, w\_initializer, b\_initializer = \

                ['eval\_net\_params', tf.GraphKeys.GLOBAL\_VARIABLES], 10, \

                tf.random\_normal\_initializer(

                    0., 0.3), tf.constant\_initializer(0.1)  # config of layers

            # first layer. collections is used later when assign to target net

            with tf.variable\_scope('l1'):

                w1 = tf.get\_variable(

                    'w1', [self.n\_features, n\_l1], initializer=w\_initializer, collections=c\_names)

                b1 = tf.get\_variable(

                    'b1', [1, n\_l1], initializer=b\_initializer, collections=c\_names)

                l1 = tf.nn.relu(tf.matmul(self.s, w1) + b1)

            # second layer. collections is used later when assign to target net

            with tf.variable\_scope('l2'):

                w2 = tf.get\_variable(

                    'w2', [n\_l1, self.n\_actions], initializer=w\_initializer, collections=c\_names)

                b2 = tf.get\_variable(

                    'b2', [1, self.n\_actions], initializer=b\_initializer, collections=c\_names)

                self.q\_eval = tf.matmul(l1, w2) + b2

        with tf.variable\_scope('loss'):

            self.loss = tf.reduce\_mean(

                tf.squared\_difference(self.q\_target, self.q\_eval))

        with tf.variable\_scope('train'):

            self.\_train\_op = tf.train.RMSPropOptimizer(

                self.lr).minimize(self.loss)

        # ------------------ build target\_net ------------------

        self.s\_ = tf.placeholder(

            tf.float32, [None, self.n\_features], name='s\_')    # input

        with tf.variable\_scope('target\_net'):

            # c\_names(collections\_names) are the collections to store variables

            c\_names = ['target\_net\_params', tf.GraphKeys.GLOBAL\_VARIABLES]

            # first layer. collections is used later when assign to target net

            with tf.variable\_scope('l1'):

                w1 = tf.get\_variable(

                    'w1', [self.n\_features, n\_l1], initializer=w\_initializer, collections=c\_names)

                b1 = tf.get\_variable(

                    'b1', [1, n\_l1], initializer=b\_initializer, collections=c\_names)

                l1 = tf.nn.relu(tf.matmul(self.s\_, w1) + b1)

            # second layer. collections is used later when assign to target net

            with tf.variable\_scope('l2'):

                w2 = tf.get\_variable(

                    'w2', [n\_l1, self.n\_actions], initializer=w\_initializer, collections=c\_names)

                b2 = tf.get\_variable(

                    'b2', [1, self.n\_actions], initializer=b\_initializer, collections=c\_names)

                self.q\_next = tf.matmul(l1, w2) + b2

    def store\_transition(self, s, a, r, s\_):

        if not hasattr(self, 'memory\_counter'):

            self.memory\_counter = 0

        transition = np.hstack((s, [a, r], s\_))

        # replace the old memory with new memory

        index = self.memory\_counter % self.memory\_size

        self.memory[index, :] = transition

        self.memory\_counter += 1

    def choose\_action(self, observation):

        # to have batch dimension when feed into tf placeholder

        observation = observation[np.newaxis, :]

        if np.random.uniform() < self.epsilon:

            # forward feed the observation and get q value for every actions

            actions\_value = self.sess.run(

                self.q\_eval, feed\_dict={self.s: observation})

            action = np.argmax(actions\_value)

        else:

            action = np.random.randint(0, self.n\_actions)

        return action

    def learn(self):

        # check to replace target parameters

        if self.learn\_step\_counter % self.replace\_target\_iter == 0:

            self.sess.run(self.replace\_target\_op)

            print('\ntarget\_params\_replaced\n')

        # sample batch memory from all memory

        if self.memory\_counter > self.memory\_size:

            sample\_index = np.random.choice(

                self.memory\_size, size=self.batch\_size)

        else:

            sample\_index = np.random.choice(

                self.memory\_counter, size=self.batch\_size)

        batch\_memory = self.memory[sample\_index, :]

        q\_next, q\_eval = self.sess.run(

            [self.q\_next, self.q\_eval],

            feed\_dict={

                self.s\_: batch\_memory[:, -self.n\_features:],  # fixed params

                self.s: batch\_memory[:, :self.n\_features],  # newest params

            })

        # change q\_target w.r.t q\_eval's action

        q\_target = q\_eval.copy()

        batch\_index = np.arange(self.batch\_size, dtype=np.int32)

        eval\_act\_index = batch\_memory[:, self.n\_features].astype(int)

        reward = batch\_memory[:, self.n\_features + 1]

        q\_target[batch\_index, eval\_act\_index] = reward + \

            self.gamma \* np.max(q\_next, axis=1)

        # train eval network

        \_, self.cost = self.sess.run([self.\_train\_op, self.loss],

                                     feed\_dict={self.s: batch\_memory[:, :self.n\_features],

                                                self.q\_target: q\_target})

        self.cost\_his.append(self.cost)

        # increasing epsilon

        self.epsilon = self.epsilon + \

            self.epsilon\_increment if self.epsilon < self.epsilon\_max else self.epsilon\_max

        self.learn\_step\_counter += 1

    def plot\_cost(self):

        import matplotlib.pyplot as plt

        plt.plot(np.arange(len(self.cost\_his)), self.cost\_his)

        plt.ylabel('Cost')

        plt.xlabel('training steps')

        plt.show()

run\_MountainCar.py

import gym

from RL\_brain import DeepQNetwork

env = gym.make('MountainCar-v0')

env = env.unwrapped

print(env.action\_space)

print(env.observation\_space)

print(env.observation\_space.high)

print(env.observation\_space.low)

RL = DeepQNetwork(n\_actions=3, n\_features=2, learning\_rate=0.001, e\_greedy=0.9,

                  replace\_target\_iter=300, memory\_size=3000,

                  e\_greedy\_increment=0.0002,)

total\_steps = 0

for i\_episode in range(10):

    observation = env.reset()

    ep\_r = 0

    while True:

        env.render()

        action = RL.choose\_action(observation)

        observation\_, reward, done, info = env.step(action)

        position, velocity = observation\_

        # the higher the better

        reward = abs(position - (-0.5))     # r in [0, 1]

        RL.store\_transition(observation, action, reward, observation\_)

        if total\_steps > 1000:

            RL.learn()

        ep\_r += reward

        if done:

            get = '| Get' if observation\_[0] >= env.unwrapped.goal\_position else '| ----'

            print('Epi: ', i\_episode,

                  get,

                  '| Ep\_r: ', round(ep\_r, 4),

                  '| Epsilon: ', round(RL.epsilon, 2))

            break

        observation = observation\_

        total\_steps += 1

RL.plot\_cost()