实验三 强化学习实验

原创

计算机的小粽子 于 2017-12-30 18:17:23 发布 6402 收藏 14

分类专栏： -------人工智能导论

本文链接：https://blog.csdn.net/tangyuanzong/article/details/78938613

实验环境选择与搭建

实验环境选择

linux + python

环境搭建

python-pip和python-dev安装

```
sudo apt-get install python-pip python-dev
```

Tensorflow安装

```
sudo pip install --upgrade https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-0.8.0-cp27-no
```

tensorflow测试

文档地址：http://wiki.jikexueyuan.com/project/tensorflow-zh/get_started/basic_usage.html

在中文文档中找个测试程序：

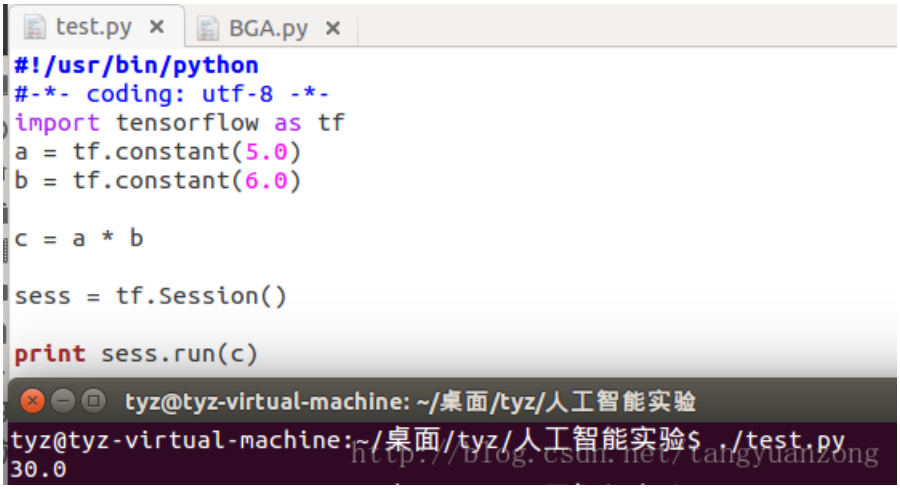
`class tf.Session`

A class for running TensorFlow operations.

A `Session` object encapsulates the environment in which `Operation` objects are executed, and `Tensor` objects are evaluated. For example:

```
# Build a graph.
a = tf.constant(5.0)
b = tf.constant(6.0)
c = a * b

# Launch the graph in a session.
sess = tf.Session()

# Evaluate the tensor `c`.
print sess.run(c)
```


测试



安装成功。。。

gym模块安装

```
sudo pip install gym[all]
```

gym环境的测试

Gym官方文档

Running an environment

Here's a bare minimum example of getting something running. This will run an instance of the `CartPole-v0` environment for 1000 timesteps, rendering the environment at each step. You should see a window pop up rendering the classic cart-pole problem:

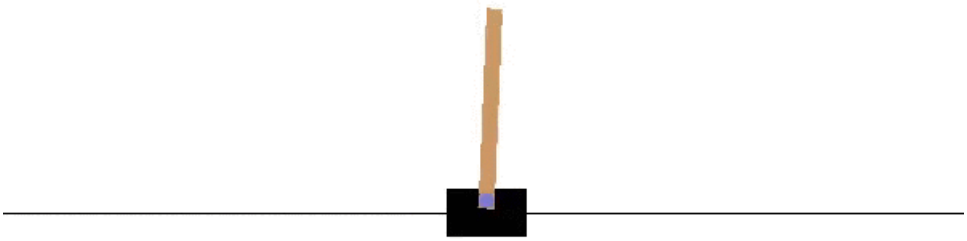
```
import gym
env = gym.make('CartPole-v0')
env.reset()
for _ in range(1000):
    env.render()
    env.step(env.action_space.sample()) # take a random action
```


测试

test.py × tensor.py ×

```
#!/usr/bin/python
# coding=utf-8
import gym
env = gym.make('CartPole-v0')
env.reset()
for _ in range(10000):
    env.render()
    env.step(env.action_space.sample())
```


得到图形

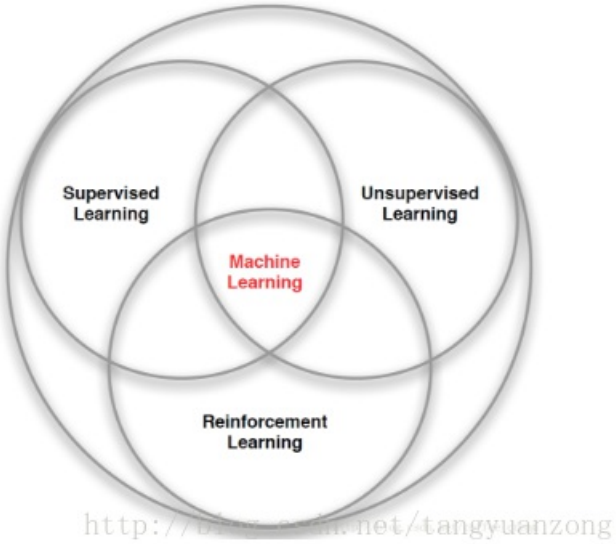


安装成功。。。

强化学习原理

机器学习分类

当前的机器学习算法可以分为3种：有监督的学习（Supervised Learning）、无监督的学习（Unsupervised Learning）和强化学习（Reinforcement Learning），结构图如下所示：



什么是强化学习？

强化学习其实也是机器学习的一个分支，但是它与我们常见的机器学习（比如监督学习supervised learning）不太一样。它讲究在一系列的情景之下，通过多步恰当的决策来达到一个目标，是一种**序列多步决策**的问题。举一个周志华老师在《机器学习》中种西瓜的例子来帮助大家理解。种瓜有很多步骤，要经过选种，定期浇水，施肥，除草，杀虫这么多操作之后最终才能收获西瓜。但是，我们往往要到最后收获西瓜之后，才知道种的瓜好不好，也就是说，我们在种瓜过程中执行的某个操作时，并不能立即获得这个操作能不能获得好瓜，仅能得到一个当前的反馈，比如瓜苗看起来更健壮了。因此我们就需要多次种瓜，不断摸索，才能总结一个好的种瓜策略。以后就用这个策略去种瓜。摸索这个策略的过程，实际上就是强化学习。可以看到强化学习有别于传统的机器学习，我们是不能立即得到标记的，而只能得到一个反馈，也可以说强化学习是一种**标记延迟的监督学习**。

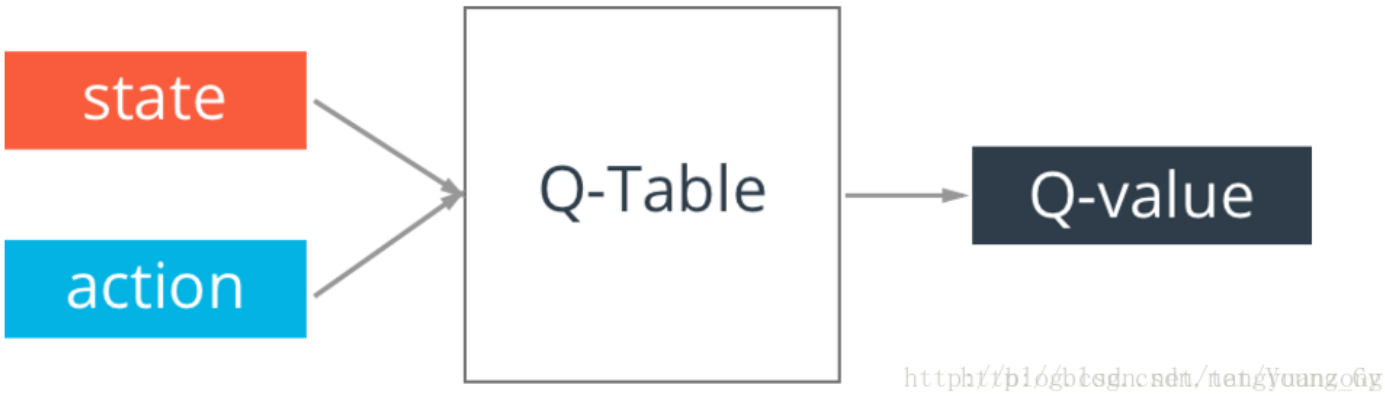
实验过程

Q-learning 算法原理介绍

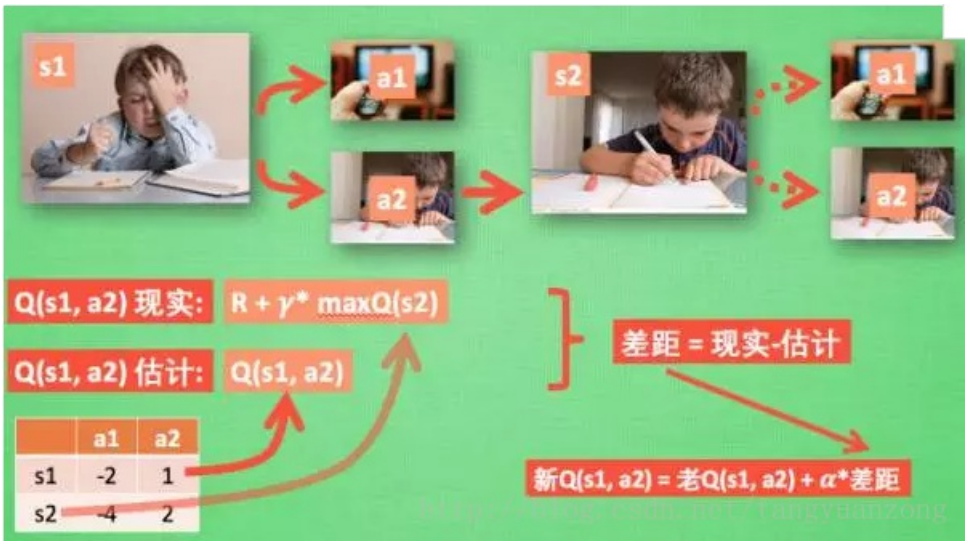
Q-learning之Q-Table

$Q(s, a)$ 衡量当前states采取actiona到底有多好。

Q-learning的核心是Q-table。Q-table的行和列分别表示state和action的值，Q-table的值

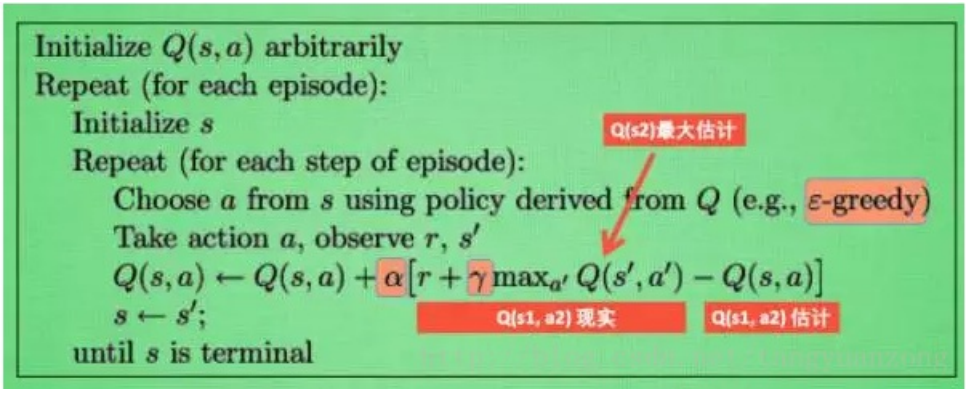


Q-Learning之Q-Table更新



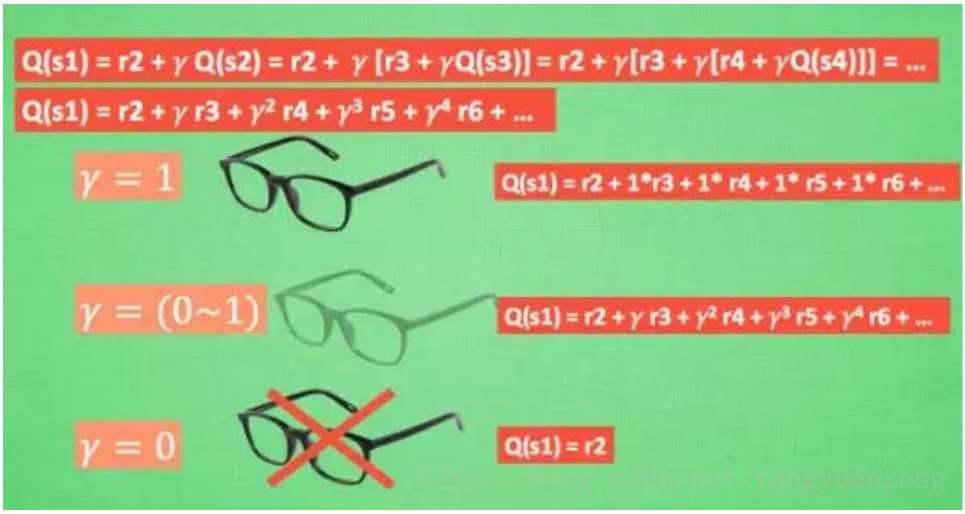
根据 Q 表的估计, 因为在 s1 中, a2 的值比较大, 通过之前的决策方法, 我们在 s1 采取了 a2, 并到达 s2, 这时我们开始更新用于决策的 Q 表, 接着我们并没有在实际中采取任何行为, 而是再想象自己在 s2 上采取了每种行为, 分别看看两种行为哪一个的 Q 值大, 比如说 Q(s2, a2) 的值比 Q(s2, a1) 的大, 所以我们把大的 Q(s2, a2) 乘上一个衰减值 gamma (比如是0.9) 并加上到达s2时所获取的奖励 R (这里还没有获取到我们的棒棒糖, 所以奖励为 0), 因为会获取实实在在的奖励 R , 我们将这个作为我现实中 Q(s1, a2) 的值, 但是我们之前是根据 Q 表估计 Q(s1, a2) 的值. 所以有了现实和估计值, 我们就能更新Q(s1, a2) , 根据 估计与现实的差距, 将这个差距乘以一个学习效率 alpha 累加上老的 Q(s1, a2) 的值 变成新的值. 但时刻记住, 我们虽然用 maxQ(s2) 估算了一下 s2 状态, 但还没有在 s2 做出任何的行为, s2 的行为决策要等到更新完了以后再重新另外做. 这就是 off-policy 的 Q learning 是如何决策和学习优化决策的过程.

Q-Learning的整体算法



每次更新我们都用到了 Q 现实和 Q 估计, 而且 Q learning 的迷人之处就是 在 Q(s1, a2) 现实 中, 也包含了一个 Q(s2) 的最大估计值, 将对下一步的衰减的最大估计和当前所得到的奖励当成这一步的现实, 很奇妙吧. 最后我们来说说这套算法中一些参数的意义. Epsilon greedy 是用在决策上的一种策略, 比如 epsilon = 0.9 时, 就说明有90% 的情况我会按照 Q 表的最优值选择行为, 10% 的时间使用随机选行为. alpha是学习率, 来决定这次的误差有多少是要被学习的, alpha是一个小于1 的数. gamma 是对未来 reward 的衰减值。

Q-Learning之Lambda



我们重写一下 Q(s1) 的公式, 将 Q(s2) 拆开, 因为Q(s2)可以像 Q(s1)一样,是关于Q(s3) 的, 所以可以写成这样, 然后以此类推, 不停地这样写下去, 最后就能写成这样, 可以看出Q(s1) 是有关于之后所有的奖励, 但这些奖励正在衰减, 离 s1 越远的状态衰减越严重. 不好理解? 行, 我们想象 Qlearning 的机器人天生近视眼, gamma = 1 时, 机器人有了一副合适的眼镜, 在 s1 看到的 Q 是未来没有任何衰变的奖励, 也就是机器人能清清楚楚地看到之后所有步的全部价值, 但是当 gamma =0, 近视机器人没了眼镜, 只能摸到眼前的 reward, 同样也就只在乎最近的大奖励, 如果 gamma 从 0 变到 1, 眼镜的度数由浅变深, 对远处的价值看得越清楚, 所以机器人渐渐变得有远见, 不仅仅只看眼前的利益, 也为自己的未来着想。

Q-Learning之决策

假设我们的行为准则已经学习好了, 现在我们处于状态s1, 我在写作业, 我有两个行为 a1, a2, 分别是看电视和写作业, 根据我的经验, 在这种 s1 状态下, a2 写作业 带来的潜在奖励要比 a1 看电视高, 这里的潜在奖励我们可以用一个有关于 s 和 a 的 Q 表格代替, 在我的记忆Q表格中, Q(s1, a1)=-2 要小于 Q(s1, a2)=1, 所以我们判断要选择 a2 作为下一个行为. 现在我们的状态更新成 s2 , 我们还是有两个同样的选择, 重复上面的过程, 在行为准则Q 表中寻找 Q(s2, a1) Q(s2, a2) 的值, 并比较他们的大小, 选取较大的一个. 接着根据 a2 我们到达 s3 并在此重复上面的决策过程. Q learning 的方法也就是这样决策的。

Q-Learning算法的通俗理解

参考某篇博客：http://blog.csdn.net/pi9nc/article/details/27649323

OpenAI Gym的使用

OpenAI Gym提供了很多测试环境，通过使用它，我们可以不用在实际中使用就能够测试算法的性能。

OpenAI Gym环境的选择

OpenAI Gym的github地址：https://github.com/openai/gym#id15

我们可以选择Toy text类型作为测试环境

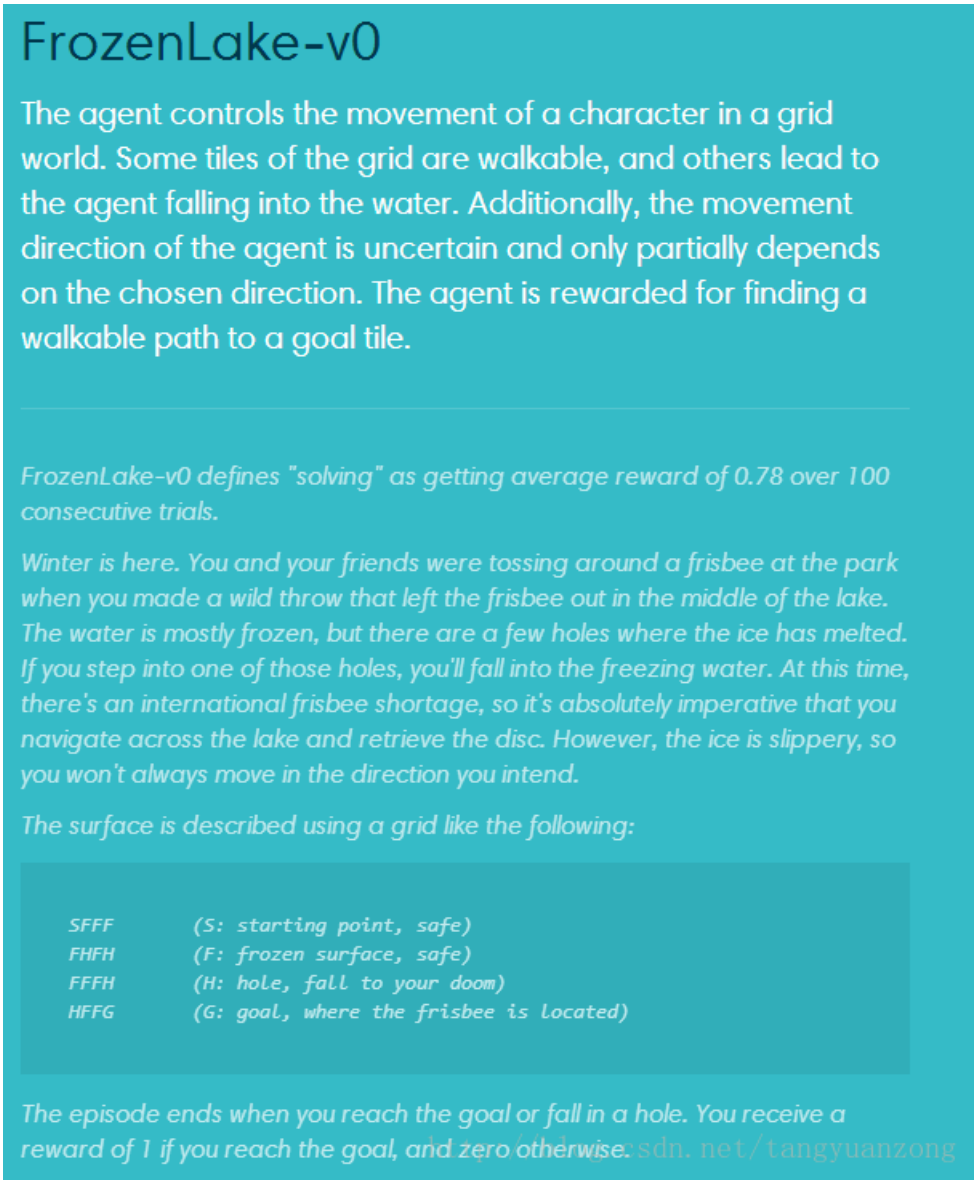
Toy text

Toy environments which are text-based. There's no extra dependency to install, so to get started, you can just do:

```
import gym
env = gym.make('FrozenLake-v0')
env.reset()
env.render()
```


Toy text的FrozenLake-v0

FrozenLake-v0是一个4*4的网络格子，每个格子可以是起始块，目标块、冻结块或者危险块。我们的目标是让agent学习从开始块如何行动到目标块上，而不是移动到危险块上。agent可以选择向上、向下、向左或者向右移动，同时游戏中还有可能吹来一阵风，将agent吹到任意的方块上。在这种情况下，每个时刻都有完美的策略是不能的，但是如何避免危险洞并且到达目标洞肯定是可行的。具体如下图：



Q-learning算法实现

奖励函数

增强学习需要我们定义奖励函数(reward function)，在我们选择的实验环境中：
Agent到达G节点，游戏结束，得到奖励1
Agent到达H节点，游戏结束，得到奖励0
Agent到达F或S节点，游戏继续，得到奖励0

代码实现：

```
    s1,r,d,_ = env.step(a)  #r就是我们采取当前action所获得的奖励
```

Q-Table

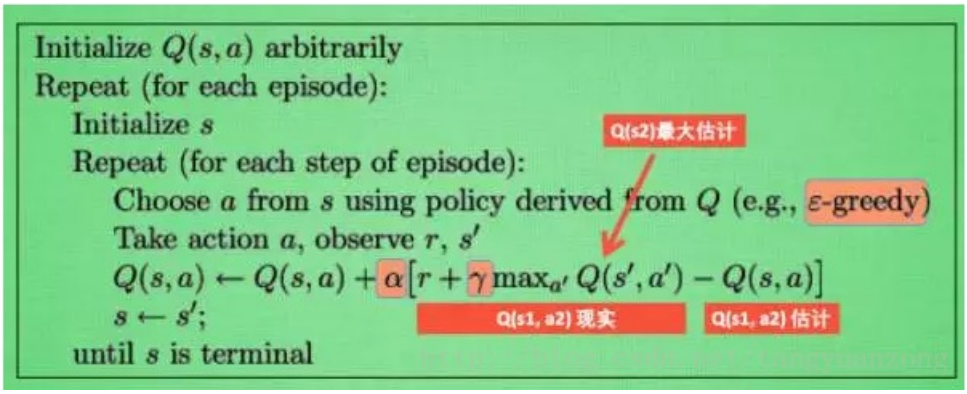
Q—Table最简单的实现方式是一个基于所有可能的状态和执行动作的查找表。在表中的每个单元格中，我们学习到在给定状态下执行特定动作的是否有效的值。在FrozenLake游戏中，我们有16种可能的状态和4种可能的动作，给出了16*4的Q值表。我们一开始初始化表格中的所有值为0，然后根据我们观察到的各种动作获得的奖励，相应地更新表格。

代码实现：

```
Q = np.zeros([env.observation_space.n,env.action_space.n])  #得到一个16*4的矩阵Q-Table，初始化为0
```

Q-learning算法代码实现

我们可以使用之前的贝尔曼方程(bellman equation)的更新来对Q表进行更新。该方程表明，给定动作的长期预期奖励来自于当前动作的即时奖励，以及来自未来最佳动作的预期奖励。




```
a = np.argmax(Q[s,:] + np.random.randn(1,env.action_space.n)*(1./(i+1))) #基于贪心法选择action
s1,r,d,_ = env.step(a)  #获取新状态和奖励，以及是否结束游戏
Q[s,a] = Q[s,a] + lr*(r + y*np.max(Q[s1,:]) - Q[s,a])  #更新Q表
s = s1
if d == True:  #如果到达G或者H
   break
```

使用Q-Table进行决策

```
while d1 == False:  #游戏继续则循环
      j1 +=1 #记录移动次数
      a = np.argmax(Q[start,:] + np.random.randn(1,env.action_space.n)*(1./(i+1))) #选取action
      s1,r,d1,_ = env.step(a) #获取新状态和奖励，以及是否结束游戏
      start = s1 #更新状态
      r_sum +=r  #获取奖励
if(r_sum == 1.0): #到达G节点
   print '达到终点，移动的次数',j1
else:
   print '未达到终点'
```

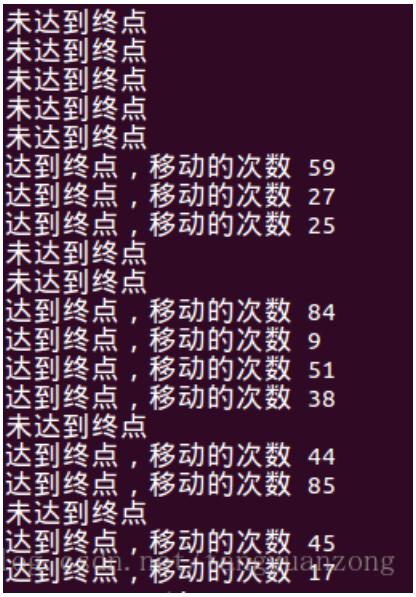
完整代码

```python
#!/usr/bin/python
# coding=utf-8
import gym
import random
import numpy as np

env = gym.make('FrozenLake-v0') #加载实验环境
print "Agent所处的环境"
env.render()  #输出4*4网格表格
rList = [] #记录奖励
lr = .85
y = .99     #权重
num_episodes = 2000  #训练次数
Q = np.zeros([env.observation_space.n,env.action_space.n]) #初始化Q表

def test(i):
    d1 = False
    j1 = 0
    start = 0
    r_sum = 0
    while d1 == False:
          j1 +=1
          a = np.argmax(Q[start,:] + np.random.randn(1,env.action_space.n)*(1./(i+1)))
          s1,r,d1,_ = env.step(a)
          start = s1
          r_sum +=r
    if(r_sum == 1.0):
       print '达到终点，移动的次数',j1
    else:
       print '未达到终点'

def Q_learning():
    for i in range(num_episodes):
        s = env.reset()  #重置环境
        rAll = 0
        d = False
        j = 0

        if (i % 1000) ==0:  #测试Q表
           test(i)

        while j < 99:  #Q-Table算法
              j+=1
              a = np.argmax(Q[s,:] + np.random.randn(1,env.action_space.n)*(1./(i+1))) #基于贪心法选择act
              s1,r,d,_ = env.step(a)  #获取新状态和奖励
              Q[s,a] = Q[s,a] + lr*(r + y*np.max(Q[s1,:]) - Q[s,a])  #更新Q表
              rAll += r
              s = s1 #更新状态
              if d == True:  #如果到达目标格
                 break

        rList.append(rAll)

Q_learning()
print "正确率: " +  str(sum(rList)/num_episodes*100) + "%"
print "得到的Q-Table"
print Q  #输出Q-Table
```

实验结果

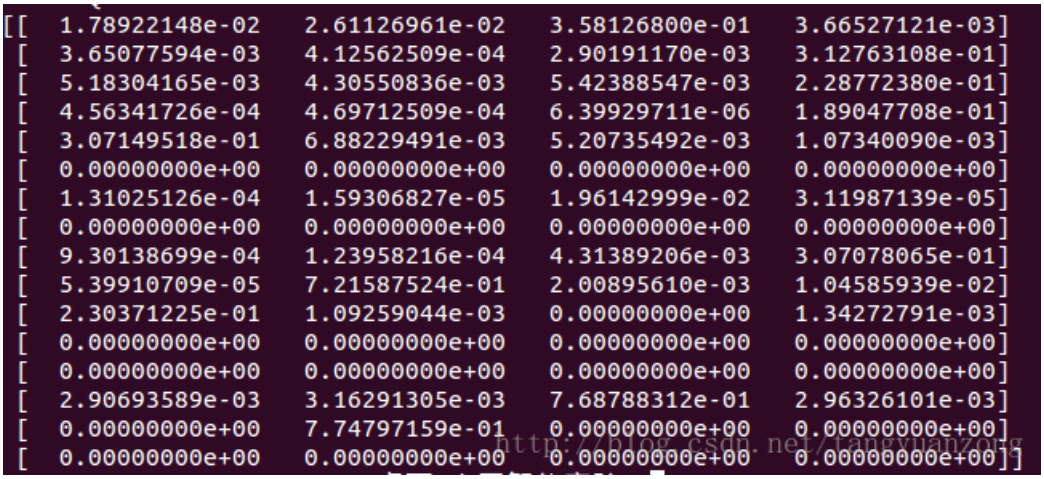
4*4网络表格

训练2000次：记录中间结果（每隔100次测试一次）



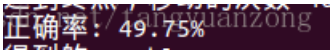
我们发现，经过多次学习。Agent找到目标点的概率增大。

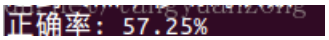
得到的Q-Table

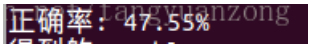


。

正确率 （求解3次）







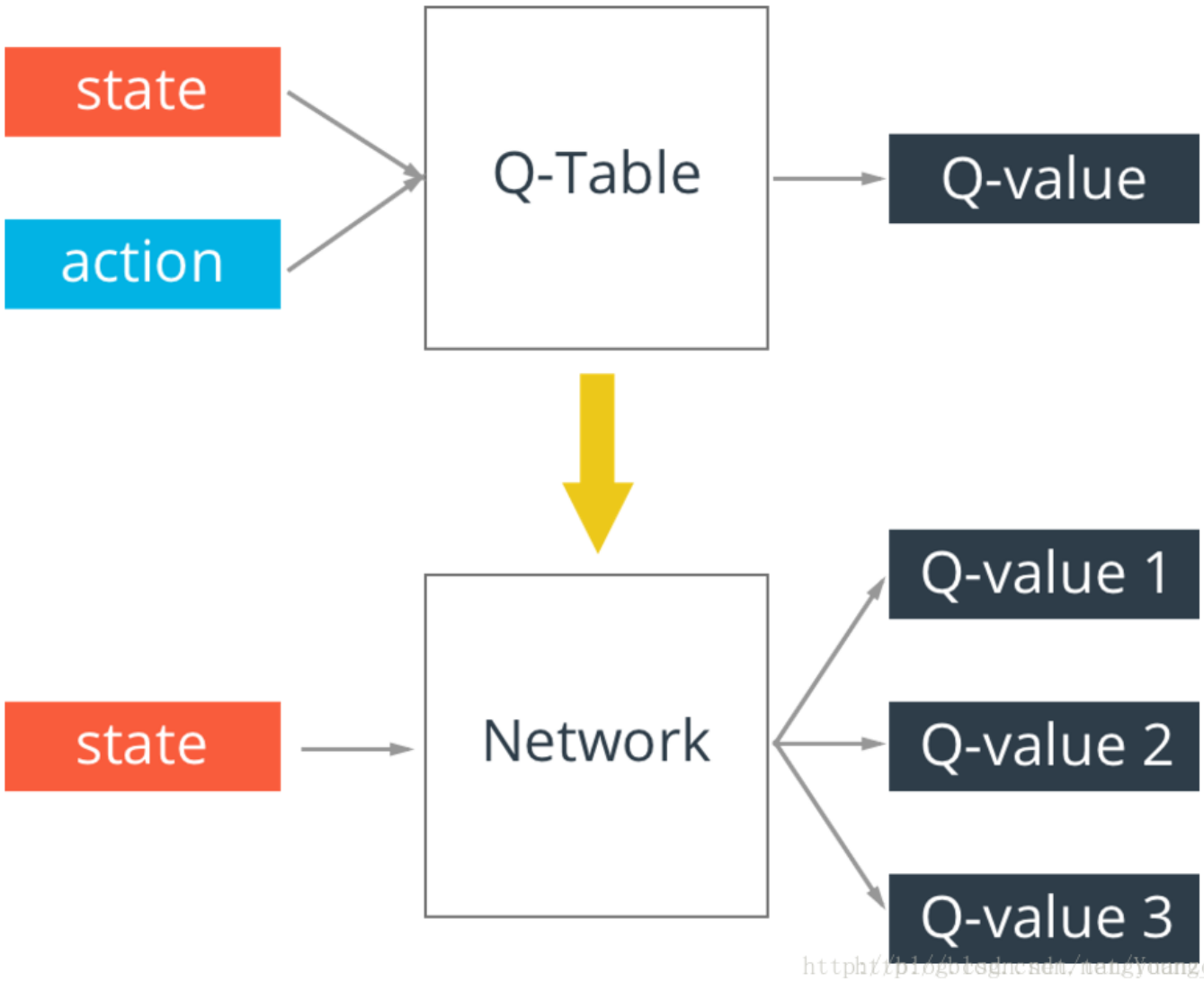
Deep-Q-learning 算法

Q-learning算法的不足

q-table方法效果不错，但是很难扩展。虽然很容易为一个简单的游戏建立16*4的表，但是在任何现代游戏或者现实世界环境中可能的状态数量几乎都是无限大的。对于大多数有趣的问题，q-table方法太简单了。所以我们需要另一种方法能够描述状态，并且生成Q值。

Deep-Q-learning

q-table存在一个问题，真实情况的state可能无穷多，这样q-table就会无限大，解决这个问题的办法是通过神经网络实现q-table。输入state，输出不同action的q-value。



误差函数

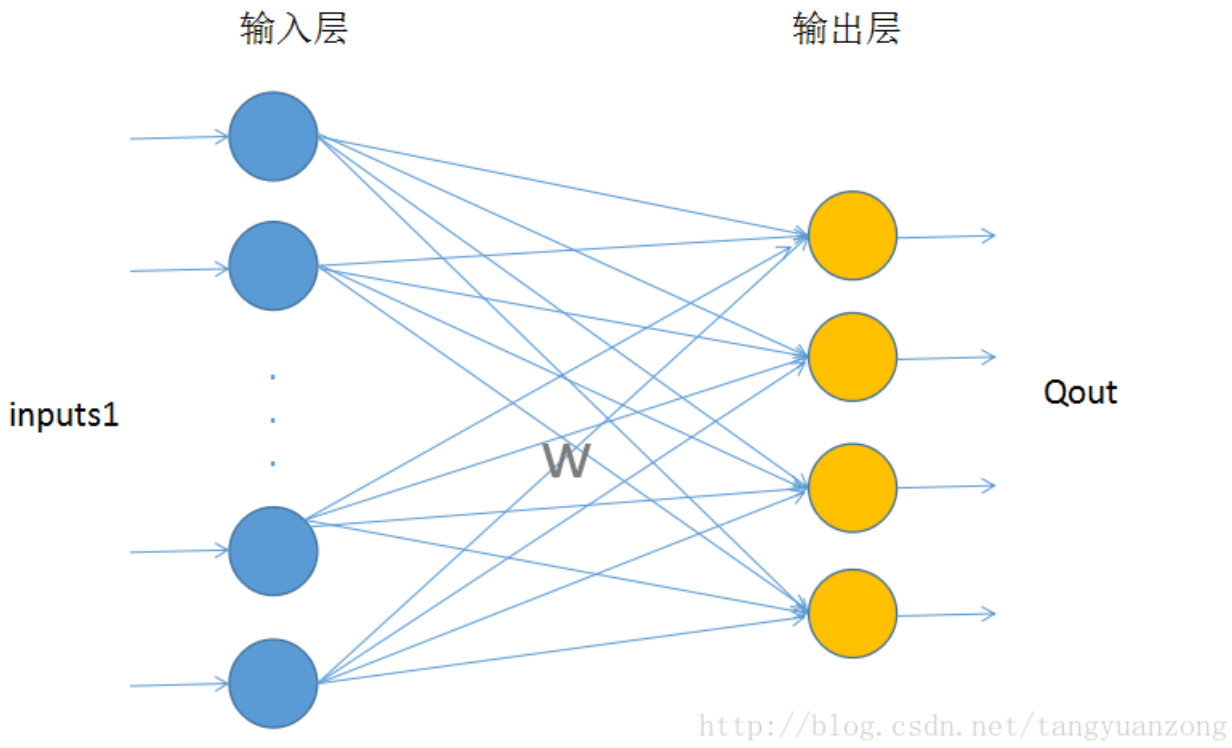
使用神经网络时，更新的方法和Q表是不同的，我们将使用反向传播算法更新损失函数。

$$Eq2.Loss = \sum(Qtarget - Q)^2$$

如果你知道BP神经网络的话，就会发现它其实就是BP神经网络。

单层网络结构

在FrozenLake例子中，我们将采用单层网络，该网络会将状态编码为独热码(one-hot vector)，并且生成一个四维的Q值向量。我们可以使用Tensorflow来选择网络的层数，激活函数和输入类型，这都上文中Q表格方法办不到的。



变量定义

输入

```
inputs1 = tf.placeholder(shape=[1,16],dtype=tf.float32) #创建输入节点(只是占位)，大小1*16的矩阵
```

权值

```
W = tf.Variable(tf.random_uniform([16,4],0,0.01)) #创建w矩阵（变量），大小为16*4，数值为0-0.01之间的float数
```

输出

```
Qout = tf.matmul(inputs1,W)  #Q网络输出结果
```

选择最优ction

```
predict = tf.argmax(Qout,1)  #取输出结果的最大值对应的action
```

误差

```
loss = tf.reduce_sum(tf.square(nextQ - Qout)) #误差
```

参数优化算法选择

本次选择梯度下降算法

```
trainer = tf.train.GradientDescentOptimizer(learning_rate=0.1) #选择梯度下降法作为优化函数
updateModel = trainer.minimize(loss)  #选择最小化loss作为优化目标
```

标准BP神经网络的缺陷

（1）容易形成局部极小值而得不到全局最优值。BP神经网络中极小值比较多，所以很容易陷入局部极小值。
（2）训练次数多使得学习效率低，收敛速度慢。
（3）隐含层的选取缺乏理论的指导。
（4）训练时学习新样本有遗忘旧样本的趋势。

算法改进

目标网络冻结(Freezing Target Networks)

Exploration：在刚开始训练的时候，为了能够看到更多可能的情况，需要对action加入一定的随机性。
Exploitation：随着训练的加深，逐渐降低随机性，也就是降低随机action出现的概率。

```
    if np.random.rand(1) < e: #引入随机性，可以避免陷入局部最优值
        a[0] = env.action_space.sample()

    if d == True: #本轮游戏结束
        e = 1./((i/50) + 10) #降低随机action出现的概率
        break
```

经验回放(Experience Replay)

强化学习由于state之间的相关性存在稳定性的问题，解决的办法是在训练的时候存储当前训练的状态到记忆体中。

```
    targetQ = allQ  #保存Q网络输出结果
```

Q网络核心代码

```
    #前向传播过程

    #生成一个16*16的单位矩阵，并且将第s行作为inputs的输入，然后计算predict和Qout，返回最优action和Q网络输出结果
    a,allQ = sess.run([predict,Qout],feed_dict={inputs1:np.identity(16)[s:s+1]})

    if np.random.rand(1) < e: #加入随机化操作，可以避免陷入局部最优值
        a[0] = env.action_space.sample()

    s1,r,d,_ = env.step(a[0]) #获取Agent新状态，奖励以及游戏状态

    #生成一个16*16的单位矩阵，并且将第s1行作为inputs的输入，然后Qout，返回Q网络输出结果
    Q1 = sess.run(Qout,feed_dict={inputs1:np.identity(16)[s1:s1+1]}) #Q1得到的是下一状态对应的Q网络输出

    maxQ1 = np.max(Q1) #从Q1选择最大的Q值
    targetQ = allQ  #保存Q网络输出结果

    targetQ[0,a[0]] = r + y*maxQ1 #更新targetQ，用来计算误差

     #反向传播过程

     #使用梯度下降法对w进行更新
    _,W1 = sess.run([updateModel,W],feed_dict={inputs1:np.identity(16)[s:s+1],nextQ:targetQ})
    rAll += r #累计奖励
    s = s1 #更新状态
    if d == True: #本轮游戏结束
        e = 1./((i/50) + 10) #减少随机值
        break
```

完整代码

```
#!/usr/bin/python
# coding=utf-8
import gym
import numpy as np
import random
import tensorflow as tf
import matplotlib.pyplot as plt

env = gym.make('FrozenLake-v0') #加载环境

tf.reset_default_graph()  #构建图表

inputs1 = tf.placeholder(shape=[1,16],dtype=tf.float32) #创建输入节点(只是占位)，大小1*16的矩阵
W = tf.Variable(tf.random_uniform([16,4],0,0.01)) #创建w矩阵（变量），大小为16*4，数值为0-0.01之间的float数

Qout = tf.matmul(inputs1,W)  #Q网络输出结果
predict = tf.argmax(Qout,1)  #取输出结果的最大值对应的action

nextQ = tf.placeholder(shape=[1,4],dtype=tf.float32) #用来计算误差(只是占位)
loss = tf.reduce_sum(tf.square(nextQ - Qout)) #误差
trainer = tf.train.GradientDescentOptimizer(learning_rate=0.1) #选择梯度下降法作为优化函数
updateModel = trainer.minimize(loss)  #选择最小化loss作为优化目标

init = tf.initialize_all_variables() #初始化所有变量

num_episodes = 2000 #训练次数

rList = [] #记录奖励
def QN():
    y = .99 # 设置学习率
    e = 0.1 #随机值
    with tf.Session() as sess: #创建sess
        sess.run(init)  #执行一次运算
        for i in range(num_episodes):

            s = env.reset() #重置环境
            rAll = 0
            d = False
            j = 0
            while j < 99:  #Q网络
                j+=1

                #前向传播过程
                #生成一个1e*16的单位矩阵，并且将第s行作为inputs的输入，然后计算predict和Qout，返回最优action和
                a,allQ = sess.run([predict,Qout],feed_dict={inputs1:np.identity(16)[s:s+1]})

                if np.random.rand(1) < e: #引入随机性，可以避免陷入局部最优值
                   a[0] = env.action_space.sample()

                s1,r,d,_ = env.step(a[0]) #获取Agent新状态，奖励以及游戏状态
                #生成一个16*16的单位矩阵，并且将第s1行作为inputs的输入，然后Qout，返回Q网络输出结果
                Q1 = sess.run(Qout,feed_dict={inputs1:np.identity(16)[s1:s1+1]}) #Q1得到的是下一状态对应

                maxQ1 = np.max(Q1) #从Q1选择最大的Q值
                targetQ = allQ  #保存Q网络输出结果

                targetQ[0,a[0]] = r + y*maxQ1 #更新targetQ，用来计算误差
```

```
                #反向传播过程
                #使用梯度下降法对W进行更新
                _,W1 = sess.run([updateModel,W],feed_dict={inputs1:np.identity(16)[s:s+1],nextQ:targe
                rAll += r
                s = s1 #更新状态
                if d == True: #游戏结束
                    e = 1./((i/50) + 10) #降低随机action出现的概率
                    break

            rList.append(rAll)

QN()
print "正确率: " +  str(sum(rList)/num_episodes*100) + "%"
```

实验结果（取三次的结果）

```
正确率: 43.0%
```

```
正确率: 8.4%
```

```
正确率: 43.55%
```

算法比较

当神经网络学习解决FrozenLake问题时，结果证明它并不像Q-table方法一样有效。虽然神经网络允许更大的灵活性，但是它们以Q-learning的稳定性为代价。Q网络的计算结果具有随机性，算法稳定性比Q-Learing差。

思考题：如何应用强化学习解决实际问题？

首先对实际问题进行抽象。然后建立模型，然后判断该使用什么机器学习算法模型。例如，该问题的取值是离散的，还是连续的。如果是离散的，可以考虑使用Q-learning算法。实际问题具有很多不确定性，我们在解决实际问题时，要考虑任何可能的随机情况。同时针对具体的问题，我们可以适当的对算法模型进行一定的修改。