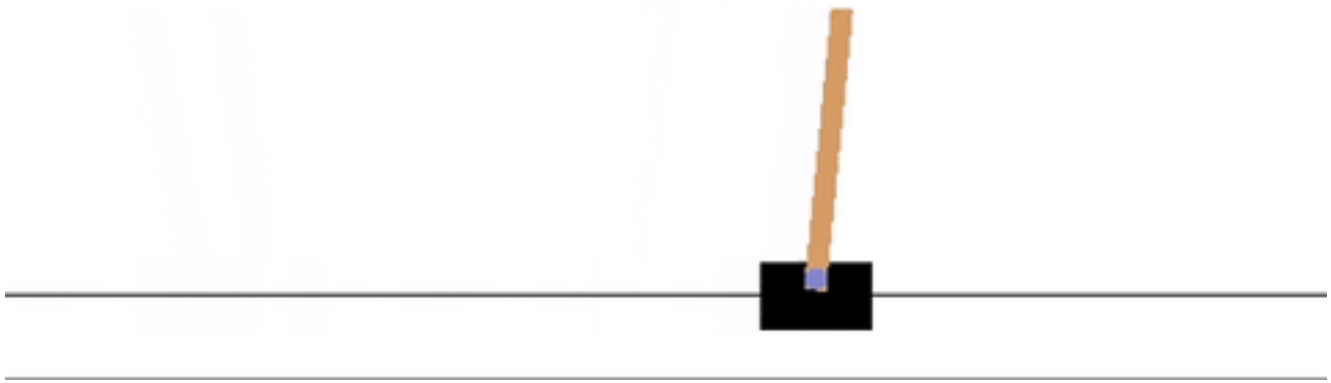


基于tensorflow的最简单的强化学习入门-part2: Policy-based Agents

felix · 6 个月前

> 本文翻译自 imple Reinforcement Learning with Tensorflow: Part 2 - Policy-based

> Agents， 作者是 Arthur Juliani， 原文链接

在Part1中，我已经介绍了如何构造一个简单的agent，该agent可以从两个不同的动作中选择回报更高的那一个。在这篇文章中，我将要基于这个agent，构造一个能够利用环境信息(observation)，并且选择能带来长远的回报的行动。有了这些，我们就有了一个完整的基于增强学习的agent。

在环境中可以获取各种信息，agent不仅仅可以得到行动后的回报(reward)和状态的变化，同时这些回报和状态是基于上一个时刻环境的状态和agent采取的行动，这样的问题称为马尔科夫问题。注意到，回报和状态可能是暂时的，并且会带有延迟。

把这个问题描述的正式一点,我们可以定义马尔科夫过程如下。 一个马尔科夫过程包括一系列可能的状态s，我们的agent在每一个时刻都会经历这些状态s，同时agent在任何时刻都可以执行动作a。 考虑状态和动作对(s,a)，新的状态s`的转移概率为T(s,a)，回报函数r为R(s,a)。如此，在马尔科夫过程的任意时刻，agent在状态s，采取了行动a，进入新的状态s`并且获得奖励r。

虽然看起来很简单，我们可以把大多数任务都看做马尔科夫过程。举个例子，想象你需要打开一扇门。我们看到的图像，身体的门的位置都可以看作状态s。身体的每一个动作都可以看作为动作a，那么奖励就是门是否被打开。同时，一直向门走是解决这个很问题很重要的过程，但是在走的过程中并不能及时的得到奖励，直到门被打开才会获得奖励。在这种情况下，agent需要学习某个动作的长久的回报。

CartPole 游戏

实现上述算法，我们需要一个比多臂老虎机更有挑战的游戏。OpenAI gym台提供了一系列增强学习的环境，在这个例子中，我们将要使用一个经典的任务，称作为CartPole，参考题图。

> 想要了解OpenAI gym平台更多的读者，可以查看这个教程。

在这个游戏中，agent将要试着学习如何平衡杠子并且保证它不会翻到，和多臂老虎机不一样，这个任务还需要满足如下条件：

- 观察(observations):神经网络需要知道杆子的状态和角度，从而输出对应动作的概率。
- 延迟奖励(Delayed reward): 保持杆子的平衡意味着agent采取动作时不仅需要考虑当前时刻，而且也需要考虑这个动作对于后续的影响。为了实现这个机制，我们需要根据时刻调整观察到的observation-action的奖励值。

考虑到时间因素，我们需要对之前教程中用到的策略梯度的形式做一定的修改。 首先在每个时刻可能需要多次更新agent。为了实现多次更新，我们需要采集agent一些列的经历(一系列回报和采取的行动)，并且在某个时刻利用这一系列的经历同时更新agent。这一系列的经历常常称为rollouts或者experience trace。

直观的，这样的策略使得每次行动不仅影响立即的回报，也可能会影响后续回报。我们现在就使用这个修改过的奖励作为我们损失方程中advantage的估计。有了这些变动，我们这就开始解决CartPole问题。

> 译者注： 代码比较复杂，建议在自己电脑上运行一下程序。

基于Policy Gradient的Tensorflow实现

如果要学习更多的强化学习算法，可以关注作者的Github

部分代码参考了Andrej Karpathy和[korymath](OpenAI Gym: korymath's algorithm on CartPole-v0

```
import numpy as np
import cPickle as pickle
import tensorflow as tf
%matplotlib inline
import matplotlib.pyplot as plt
import math
```

加载CartPole环境

```
import gym
env = gym.make('CartPole-v0')
```

首先我们试着采用随机的动作来玩这个CartPole游戏(当然结果不会太好)

```
env.reset()
random_episodes = 0
reward_sum = 0
while random_episodes < 10:
    env.render()
    observation, reward, done, _ = env.step(np.random.randint(0,2))
    reward_sum += reward
    if done:
        random_episodes += 1
        print "Reward for this episode was:",reward_sum
        reward_sum = 0
        env.reset()
```

这个任务的目标是取得200的奖励，在每一个时刻，我们都需要保持Pole的平衡，并且获得奖励。采用随机选择行动的方法，我们最多只能获得十几分，所以需要借助RL这个强大得工具取得更好效果

建立神经网络

这里建立基于Policy得神经网络，输入是观察到信息，输出左/右的概率分布。

```
# 超参数
H = 10 # number of hidden layer neurons
batch_size = 5 # every how many episodes to do a param update?
learning_rate = 1e-2 # feel free to play with this to train faster or more sta
gamma = 0.99 # discount factor for reward

D = 4 # input dimensionality

tf.reset_default_graph()

# 神经网络的输入环境的状态，并且输出左/右的概率

observations = tf.placeholder(tf.float32, [None,D] , name="input_x")
W1 = tf.get_variable("W1", shape=[D, H],
           initializer=tf.contrib.layers.xavier_initializer())
layer1 = tf.nn.relu(tf.matmul(observations,W1))
W2 = tf.get_variable("W2", shape=[H, 1],
           initializer=tf.contrib.layers.xavier_initializer())
score = tf.matmul(layer1,W2)
probability = tf.nn.sigmoid(score)
```

```
# 定义其他部分

tvars = tf.trainable_variables()
input_y = tf.placeholder(tf.float32,[None,1], name="input_y")
advantages = tf.placeholder(tf.float32,name="reward_signal")

# 定义损失函数
loglik = tf.log(input_y*(input_y - probability) + (1 - input_y)*(input_y + pro
loss = -tf.reduce_mean(loglik * advantages)
newGrads = tf.gradients(loss,tvars)

# 为了减少奖励函数中的噪声，我们累积一系列的梯度之后才会更新神经网络的参数

adam = tf.train.AdamOptimizer(learning_rate=learning_rate) # Our optimizer
W1Grad = tf.placeholder(tf.float32,name="batch_grad1") # Placeholders to send
W2Grad = tf.placeholder(tf.float32,name="batch_grad2")
batchGrad = [W1Grad,W2Grad]
updateGrads = adam.apply_gradients(zip(batchGrad,tvars))
```

优势函数(advantage function):

这个函数允许我们能够衡量agent收到的奖励。在Cart - Pole任务中，我们希望长时间保持Pole的平衡，并且能够对不好的行动有一个负的回报。由于我们需要记录一系列的奖励直到游戏结束，那么即使失败了(Pole倒了)，那么早期的行为可以看作是积极的。

```
def discount_rewards(r):
    """ take 1D float array of rewards and compute discounted reward """
    discounted_r = np.zeros_like(r)
    running_add = 0
```

```
    for t in reversed(xrange(0, r.size)):
        running_add = running_add * gamma + r[t]
        discounted_r[t] = running_add
    return discounted_r
```

训练

```
xs,hs,dlogps,drs,ys,tfps = [],[],[],[],[],[]
running_reward = None
reward_sum = 0
episode_number = 1
total_episodes = 10000
init = tf.initialize_all_variables()

# Launch the graph
with tf.Session() as sess:
    rendering = False
    sess.run(init)
    observation = env.reset() # Obtain an initial observation of the environme

    # Reset the gradient placeholder. We will collect gradients in
    # gradBuffer until we are ready to update our policy network.
    gradBuffer = sess.run(tvars)
    for ix,grad in enumerate(gradBuffer):
        gradBuffer[ix] = grad * 0

    while episode_number <= total_episodes:

        # Rendering the environment slows things down,
```

```
# so let's only look at it once our agent is doing a good job.
if reward_sum/batch_size > 100 or rendering == True :
    env.render()
    rendering = True

# Make sure the observation is in a shape the network can handle.
x = np.reshape(observation,[1,D])

# Run the policy network and get an action to take.
tfprob = sess.run(probability,feed_dict={observations: x})
action = 1 if np.random.uniform() < tfprob else 0

xs.append(x) # observation
y = 1 if action == 0 else 0 # a "fake label"
ys.append(y)

# step the environment and get new measurements
observation, reward, done, info = env.step(action)
reward_sum += reward

drs.append(reward) # record reward (has to be done after we call step(

# 批量更新
if done:
    episode_number += 1
    # stack together all inputs, hidden states, action gradients, and
    epx = np.vstack(xs)
    epy = np.vstack(ys)
    epr = np.vstack(drs)
    tfp = tfps
    xs,hs,dlogps,drs,ys,tfps = [],[],[],[],[],[] # reset array memory
```

```
            # compute the discounted reward backwards through time
            discounted_epr = discount_rewards(epr)
            # size the rewards to be unit normal (helps control the gradient e
            discounted_epr -= np.mean(discounted_epr)
            discounted_epr /= np.std(discounted_epr)

            # Get the gradient for this episode, and save it in the gradBuffer
            tGrad = sess.run(newGrads,feed_dict={observations: epx, input_y: e
            for ix,grad in enumerate(tGrad):
                gradBuffer[ix] += grad

            # If we have completed enough episodes, then update the policy net
            if episode_number % batch_size == 0:
                sess.run(updateGrads,feed_dict={W1Grad: gradBuffer[0],W2Grad:g
                for ix,grad in enumerate(gradBuffer):
                    gradBuffer[ix] = grad * 0

                # Give a summary of how well our network is doing for each bat
                running_reward = reward_sum if running_reward is None else run
                print 'Average reward for episode %f.  Total average reward %f

                if reward_sum/batch_size > 200:
                    print "Task solved in",episode_number,'episodes!'
                    break

                reward_sum = 0

            observation = env.reset()

    print episode_number,'Episodes completed.'
```

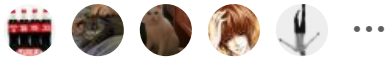
我们已经实现了一个完整的强化学习的代码，由于策略神经网络过于简单，离state-of-art还差得远呢。在下一篇文章中，我会展示如何采用深度神经网络构建agent来解决更加复杂和有趣的问题。

如果你觉得这篇文章对你有帮助，可以关注原作者。

如果你想要继续看到我的文章，也可以专注专栏。

深度学习（Deep Learning） 人工智能

强化学习 (Reinforcement Learning)



写下你的评论...

推荐阅读

JDATA京东算法大赛入门(score0.07+时间滑动窗口特征 + xgboost模型)

京东应该是第一次举办算法大赛，奖金非常诱人(虽然只是看看)。笔者看到这个比赛一激动就报名… 查看全文 >

felix · 6 个月前

基于tensorflow的最简单的强化学习入门-part1.5: 基于上下文老虎机问题(Contextual Bandits)

本文翻译自 Simple Reinforcement Learning with Tensorflow Part 1.5: Contextual Bandits… 查看全文 >

felix · 6 个月前 · 发表于 有意思的数据挖掘

他保住了摄影师的尊严，没让猴子拿到「自拍照」的著作权

2014年，有知乎er提出了这样的一个问题如果猴子拍了一张照片，那照片的版权属于谁？为什么会... 查看全文 >

波奇网 · 10 天前 · 编辑精选

承包人优先权、抵押权、消费者购房权的权利冲突及顺位

*本文经授权发布，谢绝无授权转载*当同一房屋上同时存在建设工程承包人优先权、不动产抵押权... 查看全文 >

建纬（北京）律师事务所 · 18 天前 · 编辑精选