Tensorflow_RL

<Excerpt in index | 首页摘要>
强化学习（Reinforcement Learning，简称RL）,使用Tensorflow简单学习搭建强化学习网络。
自动编码器　　GAM
Github：Tensorflow
Resource:深度学习理论与实战（基于TensorFlow实现）

<The rest of contents | 余下全文>

强化学习

1、介绍

强化学习更多的像是混合了监督学习和非监督学习，著名的 Alaph Go 就是使用强化学习的方式来学会如何下围棋，**通过不断地自我学习和对弈，战胜了人类最顶尖的棋手**，这也是强化学习的魅力所在。

在强化学习中，会有一个未知的环境，所有的电脑都能够在这个环境中**采取一些行动**，然后环境会根据不同的行动给予反馈，然后电脑再根据反馈进行操作，**不断进行。**

我们可以举一个简单的例子，比如对于超级马里奥这个游戏，我们有一个智能体，就是马里奥，那么马里奥能够**采取的行动就是前进、后退和起跳**，对于不同的动作，环境会给我们不同的反

馈，比如在一个有怪物的地方起跳，马里奥就能够躲过这个怪物，如果在这个怪物的地方选择前进，那么马里奥就会损失一条生命，得到一个负的反馈等等，**通过不断地与环境的交互，模型就会学会在不同的情况下采取合适的动作从而得到奖励，也就是说模型就能够学会一种模式，这种模式能够知道在一个环境中应该采取什么样的动作。**

在强化学习的算法中，有两个部分需要学习，**第一个部分就是环境中每一种动作的得分**，比如在马里奥的例子中，我们需要知道在怪物来到面前，我们起跳这个动作的得分。**第二个部分就是策略**，或者是说在某种特定的环境下面，我们应该采取什么样的动作来最大化得分。这其实是最像 AI 的一部分，因为这个部分包含着基于回报进行决策，非常的像人类，因为人类做的动作都是基于一些目的或者也可以称之为回报。

同时也有强化学习需要解决的两个主要内容。

第一个就是强化方式，对于一个 AI，给定一个陌生的环境，AI 必须学会如何跟环境交互来探索不同的动作能够取得怎样的奖励。同时这里也有一个权衡，就是在每一个状态下，AI 是否应该探索不同的动作从而对环境有更充分的了解，还是在每个状态只采取一个已知中最优化的动作来避免额外的风险，从而对正确的选择有了更精细的了解。所以我们需要给 AI 制定一个强化的方式来了解整个环境。

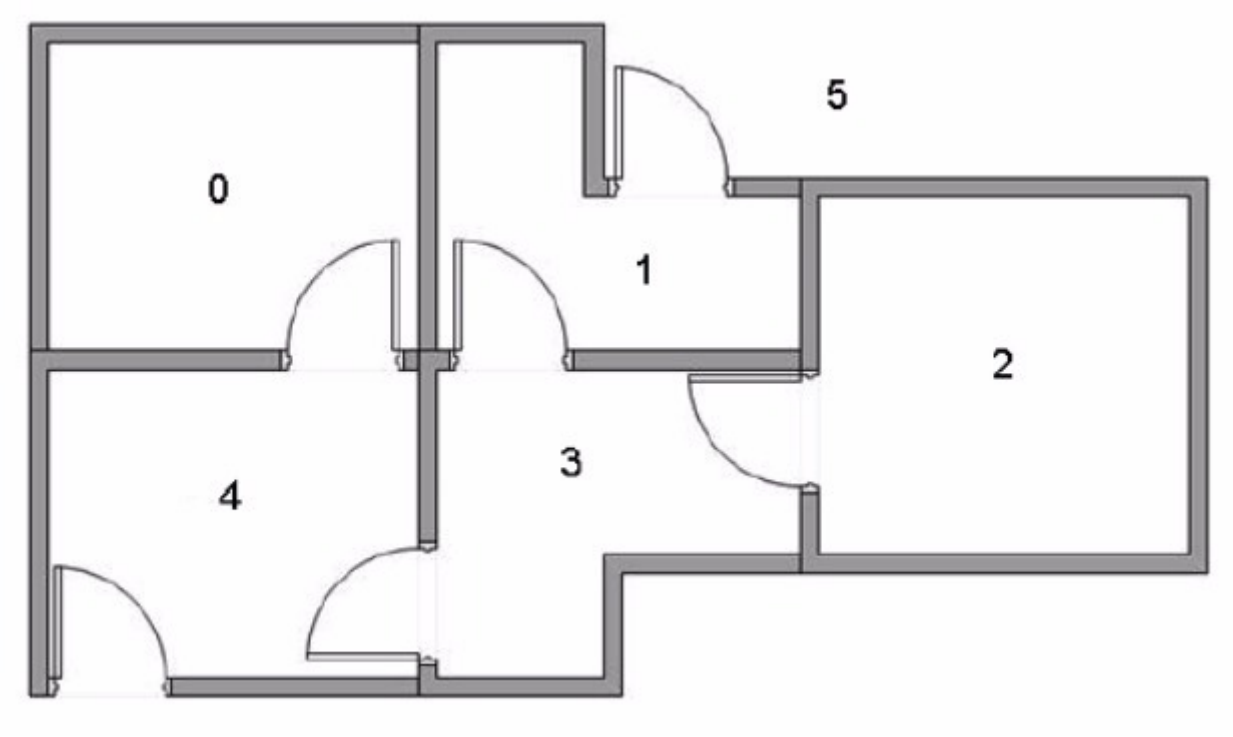
第二个就是规划，一旦对于某个环境的模型已知，我们需要知道如何根据现在的状态规划得到最大的回报。

Q Learning

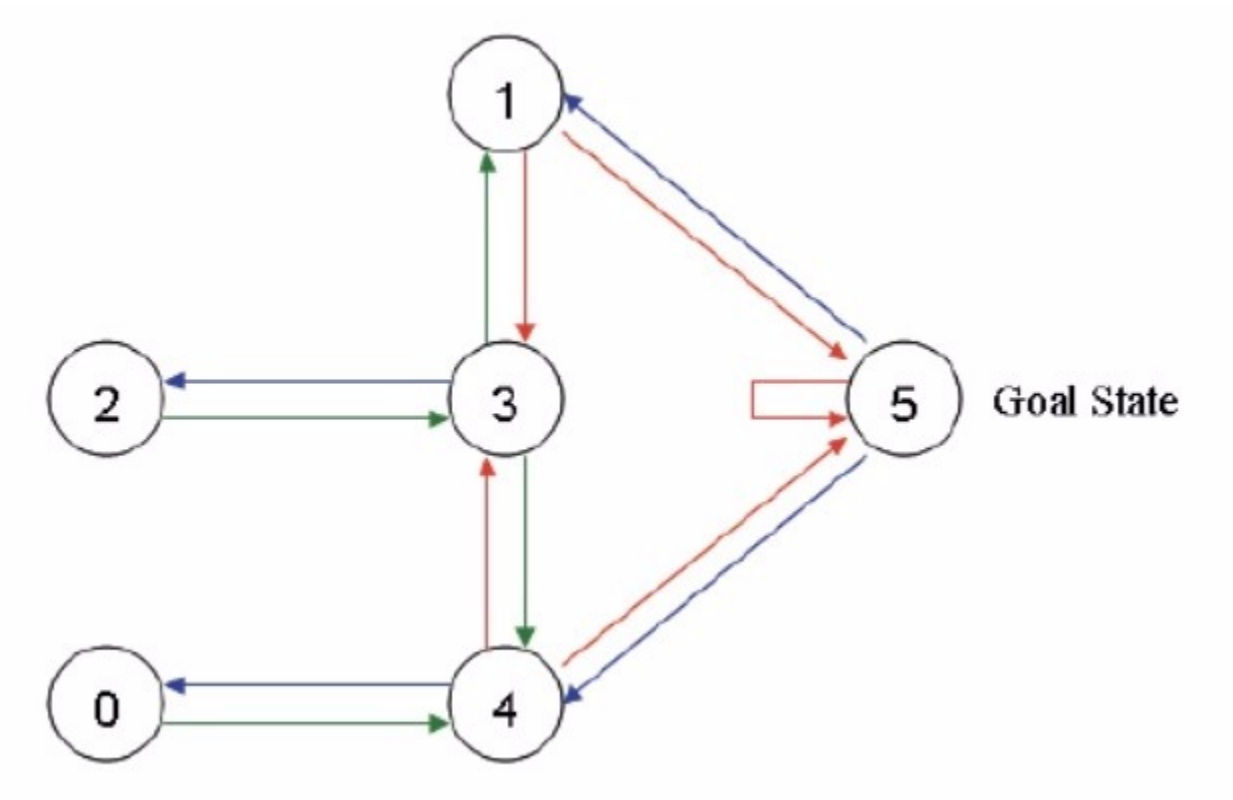
Q Learning 介绍

在增强学习中，有一种很有名的算法，叫做 **q-learning**。

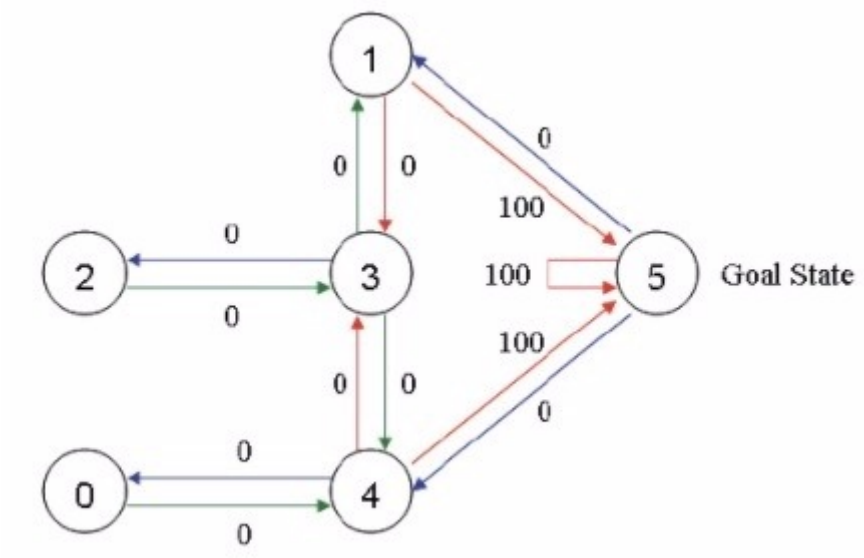
使用一个简单的例子来介绍 **q-learning**，假设一个屋子有 5 个房间，某一些房间之间相连，我们希望能够走出这个房间，示意图如下



那么我们可以将其简化成一些节点和图的形式，**每个房间作为一个节点**，两个房间有门相连，就在两个节点之间连接一条线，可以得到下面的图片



为了模拟整个过程，我们放置一个智能体在任何一个房间，希望它能够走出这个房间，也就是说希望其能够走到了 5 号节点。为了能够让智能体知道 5 号节点是目标房间，我们需要设置一些奖励，对于每一条边，我们都关联一个奖励值：**直接连到目标房间的边的奖励值设置为 100，其他的边可以设置为 0**，注意 5 号房间有一个指向自己的箭头，奖励值也设置为 100，其他直接指向 5 号房间的也设置为 100，这样当智能体到达 5 号房间之后，他就会选择一只待在 5 号房间，这也称为吸收目标，效果如下



状态和动作

q-learning 中有两个重要的概念，**一个是状态，一个是动作**，我们将**每一个房间都称为一个状态，而智能体从一个房间走到另外一个房间称为一个动作**，对应于上面的图就是每个节点是一个状态，每一个箭头都是一种行动。假如智能体处在状态 4，从状态 4 其可以选择走到状态 0，或者状态 3 或者状态 5，如果其走到了状态 3，也可以选择走到状态 2 或者状态 1 或者 状态 4。

我们可以根据状态和动作得到的奖励来建立一个奖励表，**用 -1 表示相应节点之间没有边相连，而没有到达终点的边奖励都记为 0**，如下

$$R=\begin{array}{c} \\ \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c}\begin{array}{cccccc} & & & \text{Action} & & \\ 0 & 1 & 2 & 3 & 4 & 5 \end{array}\\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix}\end{array}$$

类似的，我们可以让智能体通过和环境的交互来不断学习环境中的知识，让智能体根据每个状态来估计每种行动可能得到的收益，**这个矩阵被称为 Q 表，每一行表示状态，每一列表示不同的动作**，对于状态未知的情景，我们可以随机让智能体从任何的位置出发，然后去探索新的环境来尽可能的得到所有的状态。刚开始智能体对于环境一无所知，所以数值全部初始化为 0，如下

$$Q=\begin{array}{c}\begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array}\\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}\end{array}\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}$$

我们的智能体通过不断地学习来更新 Q 表中的结果，最后依据 Q 表中的值来做决策。

Q-learning 更新

有了奖励表和 Q 表，我们需要知道智能体是如何**通过学习来更新 Q 表**，以便最后能够根据 Q 表进行决策，这个时候就需要讲一讲 Q-learning 的算法。

Q-learning 的算法特别简单，状态转移公式如下

$$Q(s,a) = R(s,a) + \gamma \max_{\tilde{a}}\{Q(\tilde{s},\tilde{a})\}$$

其中 s, a 表示当前的状态和行动，$\tilde{s},\tilde{a}$ 分别表示 s 采取 a 的动作之后的下一个状态和该状态对应所有的行动，参数 γ 是一个常数，$0 \leq \gamma \leq 1$表示对未来奖励的一个衰减程度，**形象地比喻就是一个人对于未来的远见程度。**

解释一下就是智能体通过经验进行自主学习，不断从一个状态转移到另外一个状态进行探索，并在这个过程中不断更新 Q 表，**直到到达目标位置，Q 表就像智能体的大脑，更新越多就越强。**我们称智能体的每一次探索为 episode，**每个 episode 都表示智能体从任意初始状态到达目标状态**，当智能体到达一个目标状态，那么当前的 episode 结束，进入下一个 episode。

下面给出 q-learning 的整个算法流程

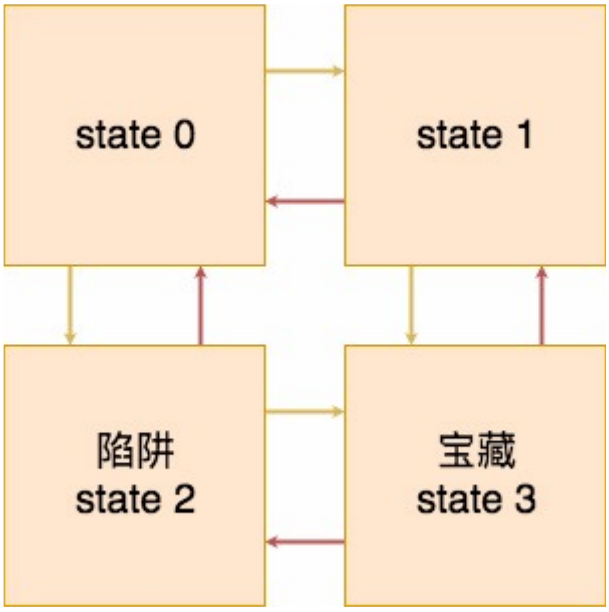
- step1 给定**参数 γ** 和**奖励矩阵 R**
- step2 令 Q:= 0
- step3 For each episode:
  - 3.1 随机选择一个**初始状态 s**
  - 3.2 若未到达目标状态，则执行以下几步
    - （1）在当前状态 s 的所有可能行动中选取一个**行为 a**
    - （2）利用选定的行为 a，得到下一个状态 $\tilde{s}$
    - （3）按照前面的转移公式计算 Q(s, a)
    - （4）令 $s := \tilde{s}$

Q Learning 简单实现

从上面的原理，我们知道了 **q-learning 最重要的状态转移公式**，这个公式也叫做 Bellman Equation，通过这个公式我们能够不断地进行更新 Q 矩阵，最后得到一个**收敛的 Q 矩阵。**

下面我们通过代码来实现这个过程

我们定义一个简单的**走迷宫过程**，也就是



最后开始让智能体与环境交互，不断地使用 bellman 方程来更新 q 矩阵，我们跑 10 个 episodeb

```
for i in range(10):
    start_state = np.random.choice([0, 1, 2], size=1)[0] # 随机初始起点
    current_state = start_state
    while current_state != 3: # 判断是否到达终点
        action = random.choice(valid_actions[current_state]) # greedy 随机
选择当前状态下的有效动作
        next_state = transition_matrix[current_state][action] # 通过选择的
动作得到下一个状态
        future_rewards = []
        for action_nxt in valid_actions[next_state]:
            future_rewards.append(q_matrix[next_state][action_nxt]) # 得到
下一个状态所有可能动作的奖励
        q_state = reward[current_state][action] + gamma * max(future_rewa
rds) # bellman equation
        q_matrix[current_state][action] = q_state # 更新 q 矩阵
        current_state = next_state # 将下一个状态变成当前状态

    print('episode: {}, q matrix: \n{}'.format(i, q_matrix))
    print()
```

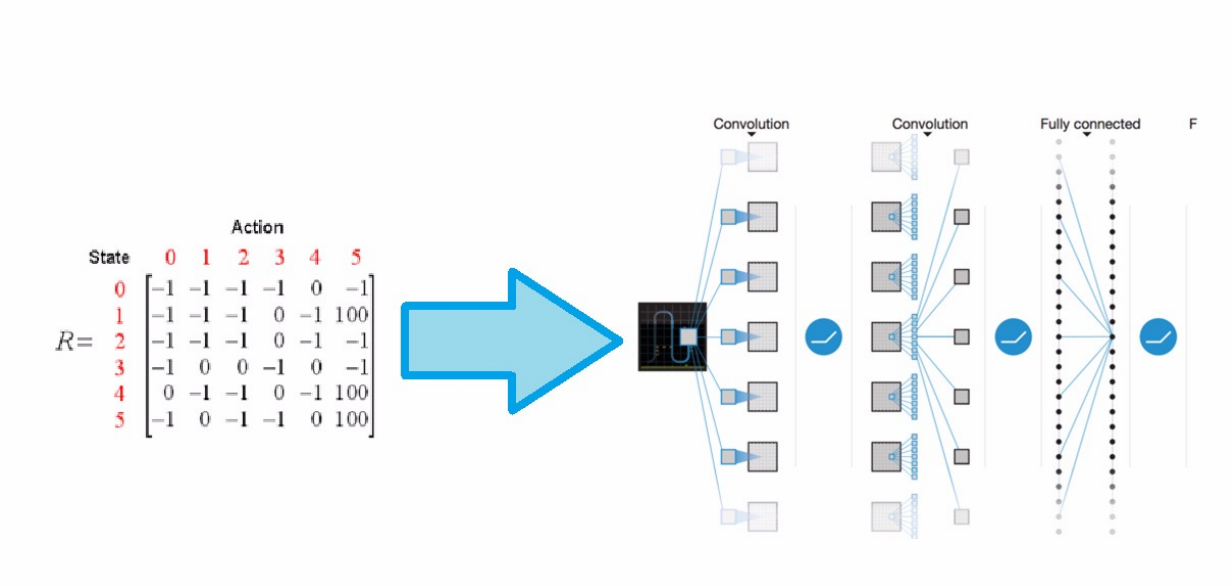
Deep Q Networks

1、Introduction

前面我们介绍了**强化学习中的 q-learning**，我们知道对于 q-learning，我们需要使用一个 Q 表来存储我们的状态和动作，每次我们使用 agent 不断探索环境来更新 Q 表，最后我们能够根据 Q 表中的状态和动作来选择最优的策略。但是使用这种方式有一个很大的局限性，如果在现实生活中，情况就会变得非常的复杂，我们可能有成千上万个 state，甚至是无穷无尽有可能是无穷无尽

的 state，对于这种情况，我们不可能将所有的 state 都用 Q 表来存储，那么我们该如何解决这个问题呢?

一个非常简单的办法就是使用深度学习来解决这个问题，所以出现了一种新的网络，叫做 **Deep Q Networks，将 Q learning 和 神经网络结合在了一起，对于每一个 state，我们都可以使用神经网络来计算对应动作的值**，就不在需要建立一张表格，而且网络更新比表格更新更有效率，获取结果也更加高效。



下面我们使用 open ai gym 环境中的 CartPole 来尝试实现一个简单的 DQN。

2、code

搭建一个二层的神经网络

```
def q_net(inputs, hidden=50, scope='q_net', reuse=None):
    """2层简单全连接神经网络
    """
    with tf.variable_scope(scope, reuse=reuse):
        # 我们用标准差为 0.1 的正态分布去初始化全连接层
        with slim.arg_scope([slim.fully_connected], weights_initializer=t
f.random_normal_initializer(stddev=0.1)):
            net = slim.fully_connected(inputs, hidden, activation_fn=tf.n
n.relu, scope='fc1')
            net = slim.fully_connected(net, n_actions,
activation_fn=None, scope='fc2')

            return net
```

LOSS

DQN 的学习过程也非常简单，我们使用 `eval net` 作为估计动作的 `value`，使用 `target net` 得到动作实际的 `value`，我们希望估计的 `value` 能够等于实际的 `value`，所以可以使用 mse 来作为 loss 函数就可以了。

```
with tf.variable_scope('loss'):
    loss = tf.reduce_mean(tf.losses.mean_squared_error(q_target, q_eval))
with tf.variable_scope('train'):
    train_op = tf.train.RMSPropOptimizer(lr).minimize(loss)
```

具体实现参考GitHub中的`deep Q learning`文件

Gym 介绍

前面我们简单的介绍了强化学习的例子，从这个例子可以发现，**构建强化学习的环境非常麻烦**，需要耗费我们大量的时间，这个时候我们可以使用一个`开源的工具，叫做 gym`，是由`open ai`开发的。

在这个库中从简单的走格子到毁灭战士，提供了**各种各样的游戏环境可以让大家放自己的 AI 进去玩耍**。取名叫 gym 也很有意思，可以想象一群 AI 在健身房里各种锻炼，磨练技术。

使用起来也非常方便，首先在终端内输入如下代码进行安装。

```
# Github源
git clone https://github.com/openai/gym
cd gym
pip install -e .[all]

# 直接下载gym包
pip install gym[all]
```

我们可以访问这个页面看到`gym`所包含的环境和介绍。

反馈与建议

- 微博：@柏林designer
- 邮箱：wwj123@zju.edu.cn