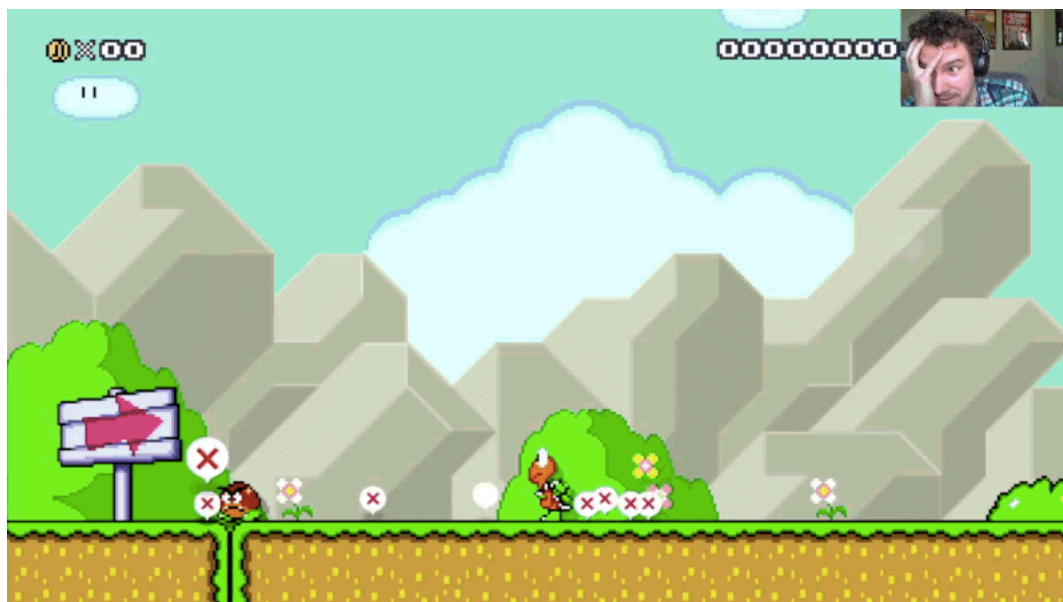


强化学习介绍

前面我们学习了两种机器学习和深度学习中的主流，监督学习，比如卷积网络用于图像分类，非监督学习，比如自动编码器和生成对抗网络。下面我们会讲一种更加智能化的学习方式，这种方式更多的像是混合了监督学习和非监督学习，叫做强化学习或者增强学习，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行动。

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，或者说是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。

上面这两个问题就组成了强化学习的主要内容，也是强化学习不断在解决的东西。

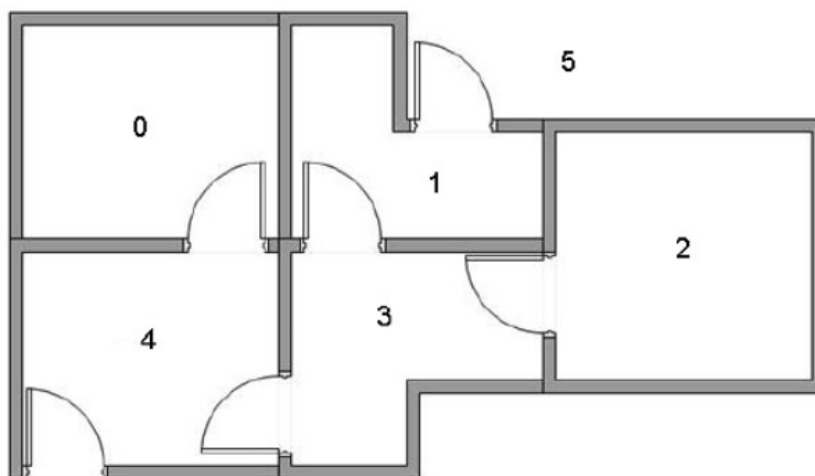
上面我们介绍了强化学习的基本内容，下一章我们将逐渐对强化学习中的几个经典算法慢慢展开学习。

Q Learning 介绍

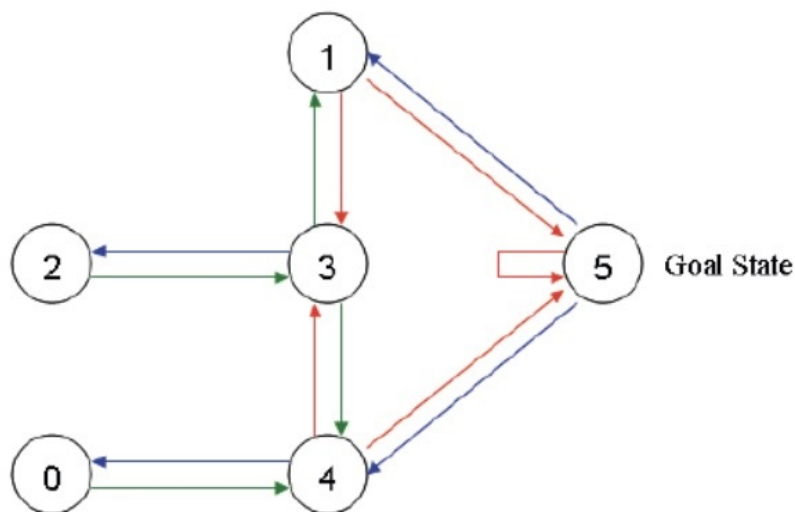
在增强学习中，有一种很有名的算法，叫做 q-learning，我们下面会从原理入手，然后通过一个简单的小例子讲一讲 q-learning。

q-learning 的原理

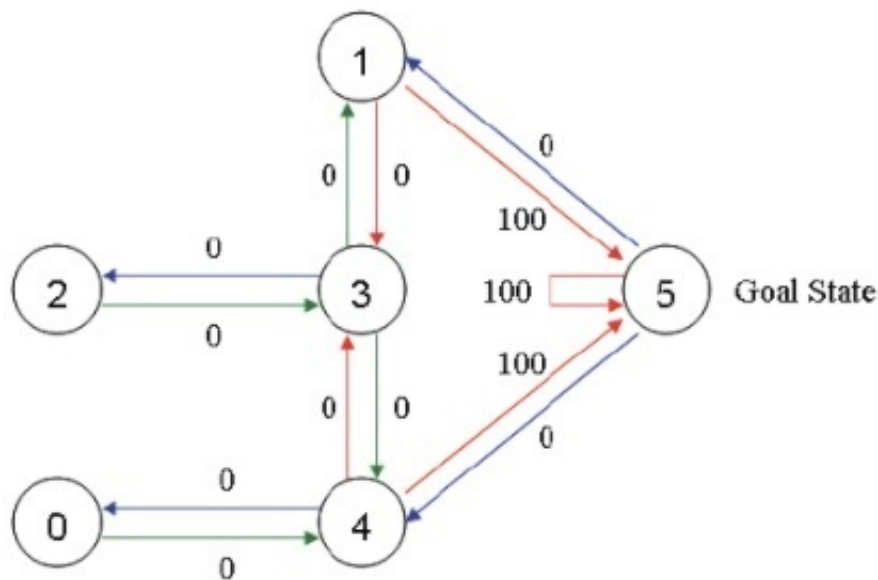
我们使用一个简单的例子来导入 q-learning，假设一个屋子有 5 个房间，某一些房间之间相连，我们希望能够走出这个房间，示意图如下



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



想想一下智能体可以不断学习，每次我们将其放在其中一个房间，然后它可以不断探索，根据奖励值走到 5 号房间，也就是走出这个屋子。比如现在这个智能体在 2 号房间，我们希望其能够不断探索走到 5 号房间。

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

		Action					
State		0	1	2	3	4	5
$R =$	0	-1	-1	-1	-1	0	-1
	1	-1	-1	-1	0	-1	100
	2	-1	-1	-1	0	-1	-1
	3	-1	0	0	-1	0	-1
	4	0	-1	-1	0	-1	100
	5	-1	0	-1	-1	0	100

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

		0	1	2	3	4	5
$Q =$	0	0	0	0	0	0	0
	1	0	0	0	0	0	0
	2	0	0	0	0	0	0
	3	0	0	0	0	0	0
	4	0	0	0	0	0	0
	5	0	0	0	0	0	0

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 算法

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 $s := \tilde{s}$

单步演示

为了更好地理解 q-learning，我们可以示例其中一步。

首先选择 $\gamma = 0.8$ ，初始状态为 1， Q 初始化为零矩阵

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

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

因为是状态 1，所以我们观察 R 矩阵的第二行，负数表示非法行为，所以下一个状态只有两种可能，走到状态 3 或者走到状态 5，随机地，我们可以选择走到状态 5。

当我们走到状态 5 之后，会发生什么事情呢？观察 R 矩阵的第 6 行可以发现，其对应于三个可能采取的动作：转至状态 1，4 或者 5，根据上面的转移公式，我们有

$$Q(1, 5) = R(1, 5) + 0.8 * \max\{Q(5, 1), Q(5, 4), Q(5, 5)\} = 100 + 0.8 * \max\{0, 0, 0\} = 100$$

所以现在 Q 矩阵进行了更新，变为了

$$Q = \begin{matrix} & \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{matrix}$$

现在我们的状态由 1 变成了 5，因为 5 是最终的目标状态，所以一次 episode 便完成了，进入下一个 episode。

在下一个 episode 中又随机选择一个初始状态开始，不断更新 Q 矩阵，在经过了很多个 episode 之后，矩阵 Q 接近收敛，那么我们的智能体就学会了从任意状态转移到目标状态的最优路径。

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(<https://github.com/openai/gym/wiki>)。

在上面的环境页面，可以 gym 内置了很多环境，我们可以使用前面讲过的 q learning 尝试一个 gym 中的小例子，mountain car，<https://github.com/openai/gym/wiki/MountainCar-v0>。在 mountain car，我们能够观察到环境中小车的位置，也就是坐标，我们能够采取的动作是向左或者向右。

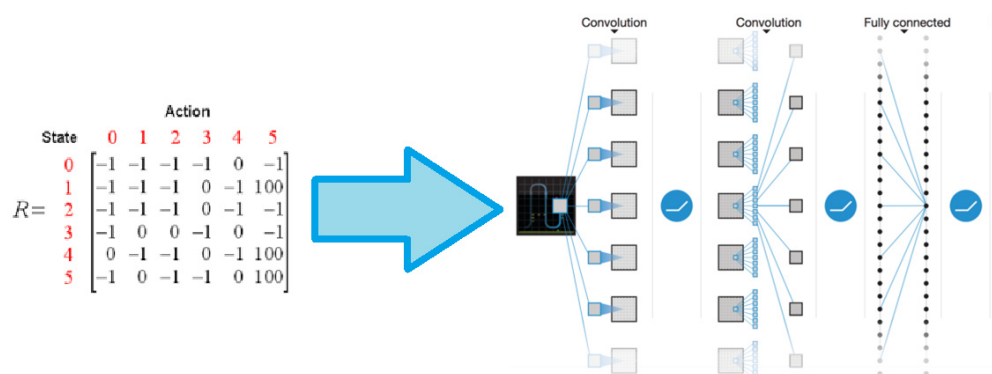
为了使用 q learning，我们必须要建立 q 表，而这里的状态空间是连续不可数的，所以我们需要离散化连续空间，将 x 坐标和 y 坐标都平均分成很多份，具体的实现可以运行 `mount-car.py` 看看结果。

如果运行完之后，可以看到 q 表的收敛非常慢，reward 一直都很难变化，我们需要很久才能将小车推到终点，这个时候我们需要一个更加强大的武器，那就 deep q network。

Deep Q Networks

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。



更新方法

dqn 的更新过程和之前讲个 q learning 的过程是很相同的

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function \hat{Q} with weights $\theta^- = \theta$

For episode = 1, M **do**

 Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

For $t = 1, T$ **do**

 With probability ε select a random action a_t

 otherwise select $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

 Execute action a_t in emulator and observe reward r_t and image x_{t+1}

 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

 Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$

 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ

 Every C steps reset $\hat{Q} = Q$

End For

End For

上图中有两个网络，第一个网络叫做 action-value function Q ，还有一个网络叫做 target action-value \hat{Q} ，第一个网络可以称为 eval net，第二个网络叫做 target net，eval net 每一步都更新参数，而 target net 每过 C 步才更新一次，更新地方式是直接将 eval net 的参数复制到 target net 上，为什么要学习两个网络而不学习一个网络呢？因为 DQN 希望每次更新能够有一定的时间去探索周围的环境而得到更多的信息再来更新，在游戏中的过程是一个时间序列，所以得到的信息是有着时间依赖关系，每次都直接使用序列信息进行更新并不能得到好的结果，所以我们就采用滞后更新的方式，每隔 C 步才真正更新一次参数。

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，所以可以使用均方误差来作为 loss 函数就可以了。