Lecture 10：Introduction to Deep Reinforcement Learning（RL）

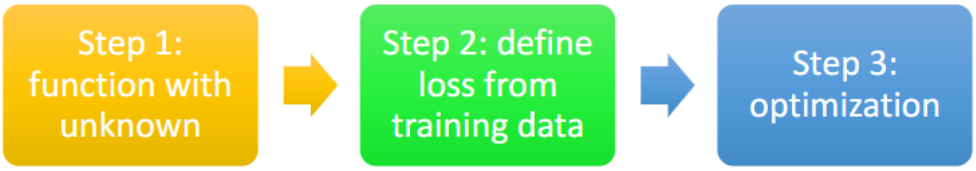
Lectured by HUNG-YI LEE (李宏毅)
Recorded by Yusheng Zhao（yszhao0717@gmail.com）

之前的Lecture基本都是基于Supervised Learning，supervised learning通常基于labelled的数据，但是有些场景，譬如下围棋玩游戏，对于下一步的决策如何是未知的（假设machine是尝试无师自通，不看棋谱）。当我们收集有label的资料很困难的时候，我们可能就需要考虑RL。RL的方法下，机器会和环境做互动，会得到奖励，一般称之为reward...（actor就是agent，可能有不同表述）

What is RL？（3 steps in ML）

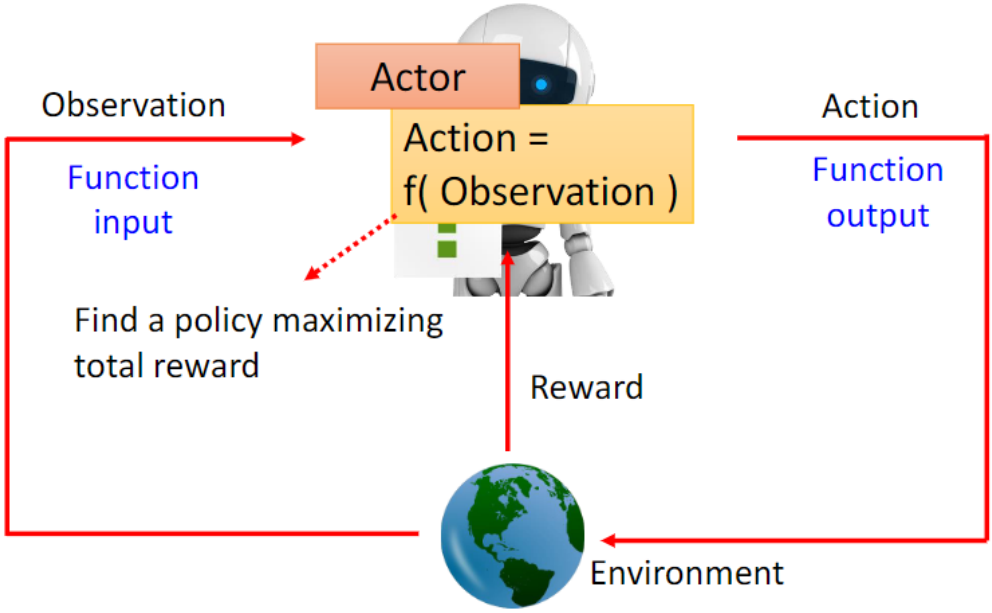
> 通常的介绍RL课程会从马尔可夫决策过程 (**Markov Decision Process**，MDP)引入。这里稍有不同，从ML类比引入。

实际上，RL和ML是一样的框架（three steps）。



Step 1：Function with Unknown

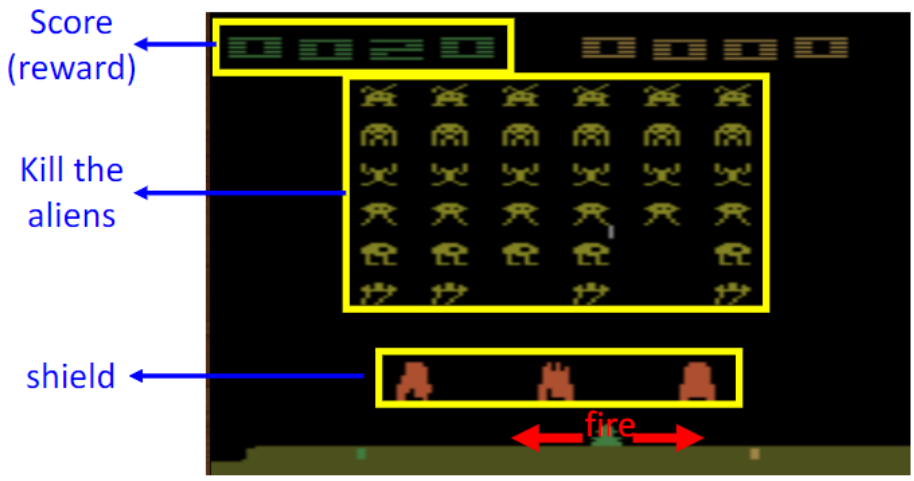
RL里面会有一个**Actor**以及一个**Environment**，两者会产生互动。environment会给actor一个**observation**（作为actor的输入）；之后，actor会有一个输出称之为**action**，action会影响Environment，从而给出新observation...



Actor就是我们要找的"function"：*Action = f(Observation)*，输入就是环境给的observation，输出就是这个actor要采取的action。在互动的过程，这个environment会不断给这个actor一些**reward**——让actor知道其所输出的action的好坏。

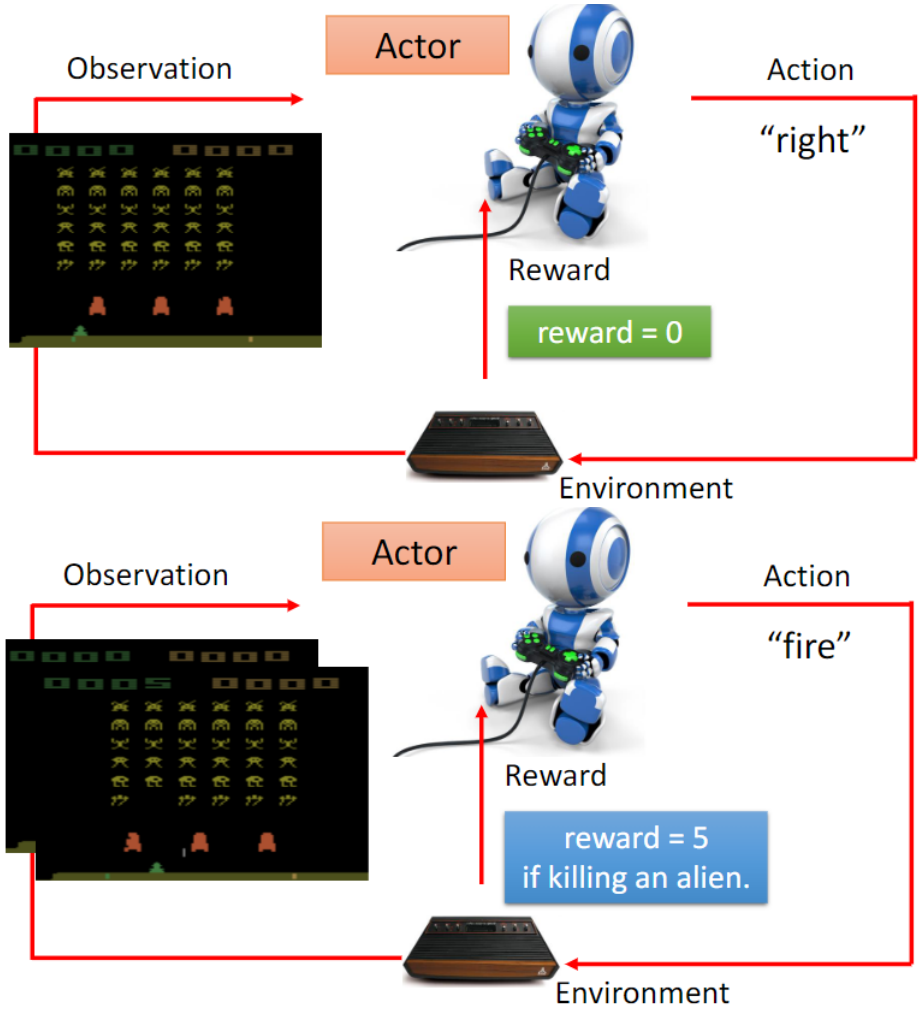
这个function（即actor）的目标就是去maximize从environment所获得的reward的总和。

Atari游戏为例——Space invader



· Actor：玩家；
· Observation：游戏状况（Aliens & shields）
· Action：左移、右移、开火
· Reward：Score（只有杀掉外星人才会得到分数）
· 终止条件（Termination）：aliens被杀光或者你的飞船被击杀

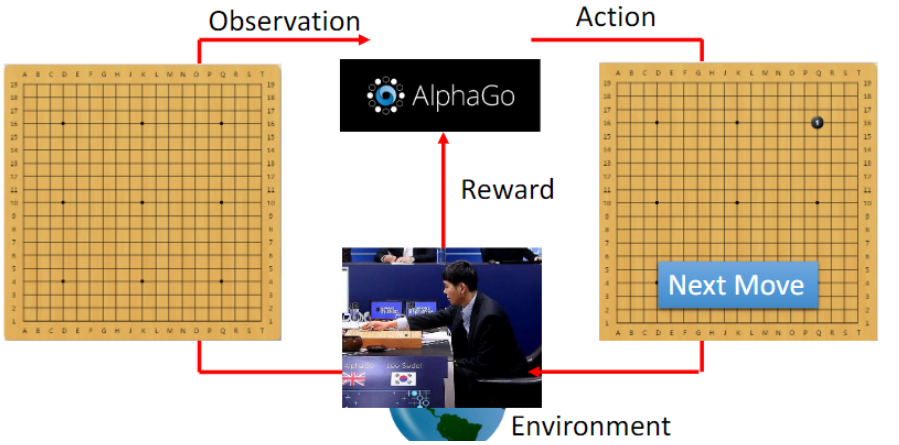
将上述游戏抽象为Actor-Environment交互关系，如下图，当action为移动时，reward为0；当action开火击杀alien，则score+5（reward增加）。每采取一个action，都会让environment更新一个observation给actor作为输入。

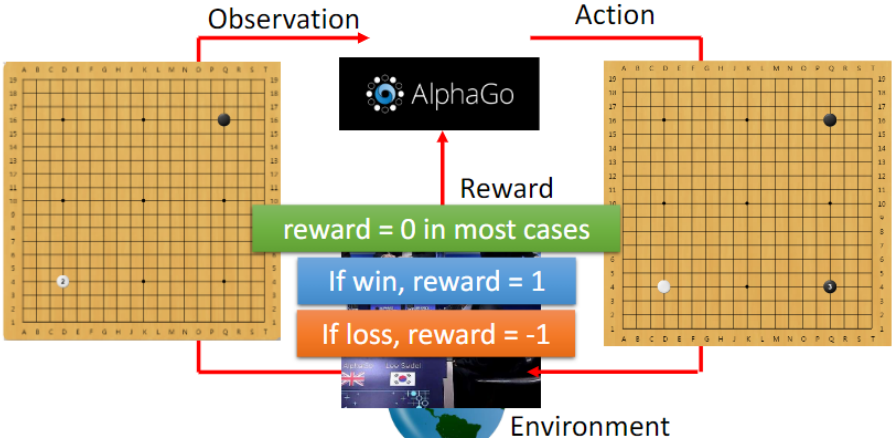


我们要求这个actor要不断朝向"获得最多分数"的方向行进。（reward maximization）

围棋为例

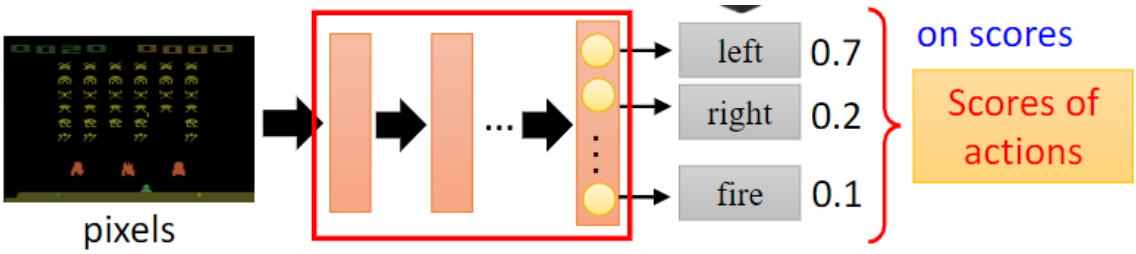
Actor就是"AlphaGo"，Environment就是"人类对手"，observation就是"棋盘的局势"。action就是"程序下一步落子何处"（19×19），实际上输出也是棋盘。





和Atari游戏不同的是，在下围棋的整个过程中，只有游戏结束Actor才能拿到reward，actor无法得到实时的reward的反馈。

这个未知数的function（Function with Unknown）就是Actor，Actor目前通常称之为**Policy Network**。在RL不用神经网络技术之前，通常这时候的actor比较简单，可能就是一个look-up table。在上述实例的Atari游戏中，



输入就是整个游戏场景（图像），通过整个复杂的NN（Actor），给action做一个分数量化（如上图所示）。这个network事实上和分类任务的NN是一样的。如何设计网络的架构取决于我们自己。对于输入是一张图片的environment，我们可能会想到用CNN做这个网络；甚至，如果我们需要一个长序列的游戏画面（若干帧），那么就可以用RNN或者transformer。

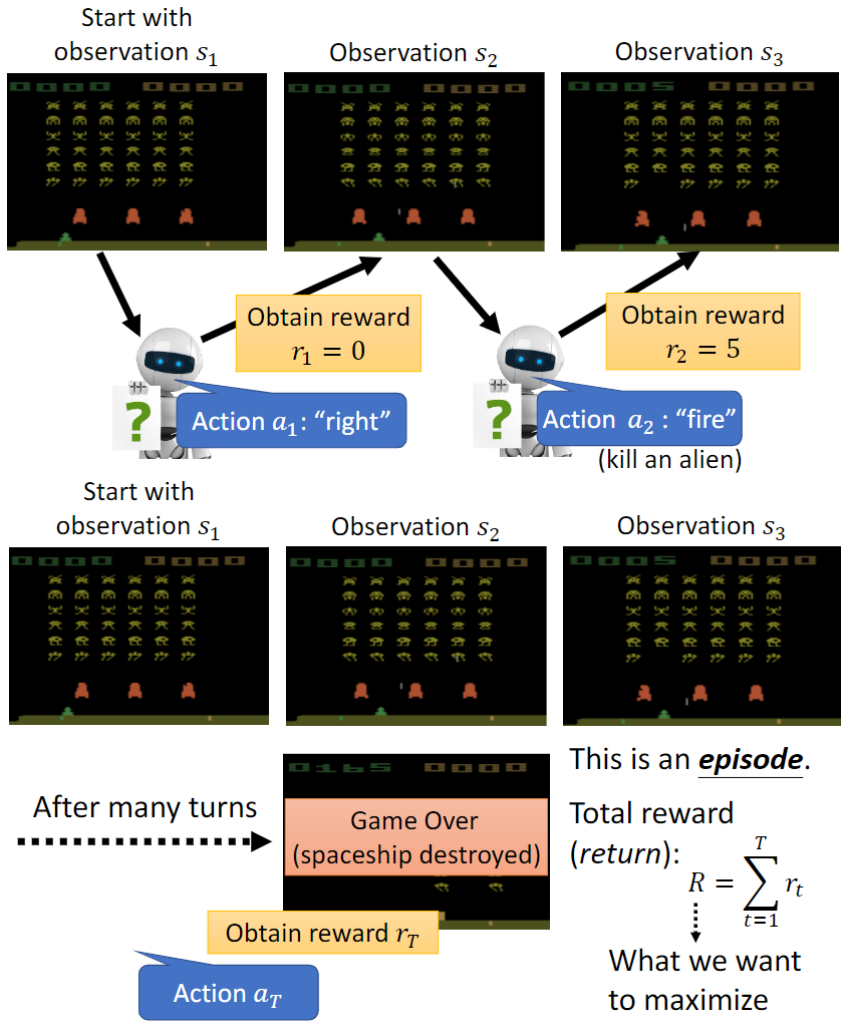
> • Input of neural network: the observation of machine represented as a vector or a matrix
> • Output neural network : each action corresponds to a neuron in output layer

network的架构有我们自行设计，对于以上这种任务，我们只要输入是画面，输出是类似于类别的action就行。actor会采用哪一个action取决于action中各类别的分数，常见的做法是把这个分数当作一个几率（如上图就是0.7，0.2，0.1）然后直接sample到选择某个action。多数RL都是用sample的方法（当然你可以确定性的选择分数最大的那个），sample方法的好处在于即便在相同的任务场景（输入）下，其输出的选择action也可能略有不同。随机性可以让模型更鲁棒。~~联想玩石头剪子布的小时当，雾:）~~

这些unknown的东西就是Policy Network中的参数，也是我们需要train出来的东西。

Step 2：Define "Loss"

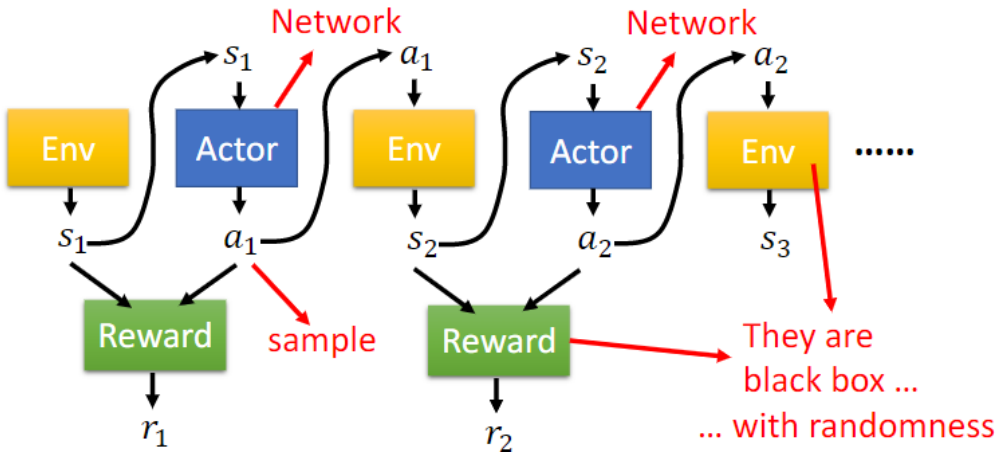
actor和environment交互的场景：初始的游戏画面作为observation s_1让actor产生一个输出action a_1，从environment获得reward r_1，这致使environment产生新的observation s_2，其让actor产生输出a_2，从environment中获得r_2...获得r_T...周而复始该过程，直到机器采取某个action（达成某种游戏结束的条件），game over。



从游戏开始到结束整一个过程就称之为***episode***，一个episode会得到一个*total reward*：$R = \sum_{t=1}^{T} r_t$，在一些文献中，这个total reward也称之为*return*。ok，我们决定了这个优化目标——maximize the total reward（or minimize the minus total reward）$Loss\ Function \sim -R$

Step 3：Optimization

Actor-Env交互流程如下



我们把序列$\tau = \{s_1, a_1, s_2, a_2, \ldots\}$称之为**Trajectory**。Reward（作为一个function）其输出同时取决于action和observation，即$r_i = R(s_i, a_i)$；这个问题的优化目标可以写为

$$R(\tau) = \sum_{t=1}^{T} r_t \tag{1}$$

整个optimization的问题可以描述为找一个network（或者train出一组参数）放在actor位置，可以让R的值越大越好。

> 如果环境是已知的，reward就是已知的；反之，reward也属于black-box function，或许也需要被train出来。

关键在于，RL和一般的ML有一些不同（这也造成了RL做optimization的困难）

- Actor的输出是有随机性的，例如对于同样的s_1，sample出来的a_1可能不一样
- Environment和Reward都是black-box的，根本不知道里面发生了什么，对于外界而言这部分就是end-to-end的...
- Environment或者Reward都可能也具备随机性

RL的难点就是如何解决这个optimization的问题，如何设计自动化方法来maximize这个R

类比GAN（异曲同工之妙，有相似，也有不一样的地方），Actor就像是Generator，Env和Reward就是Discriminator，优化目标就是调整Generator的参数让Discriminator越大越好。不一样的地方在于，GAN里面Discriminator也是一个NN（我们可以GD来train），而Env和Reward是黑盒子而不是network，不能用梯度下降方法来train它。

> $-R = Loss$，优化目标就是让$-R$越小越好。
>
> 在之前的Deep Learning中Random_Seed的随机性体现在training当中，每次train的时候init的参数会不一样，而在RL中所谓Actor在testing当中就已经有随机性了。（我们拿train好的actor去测试环境，即便相同的输入通常会用不同的输出）

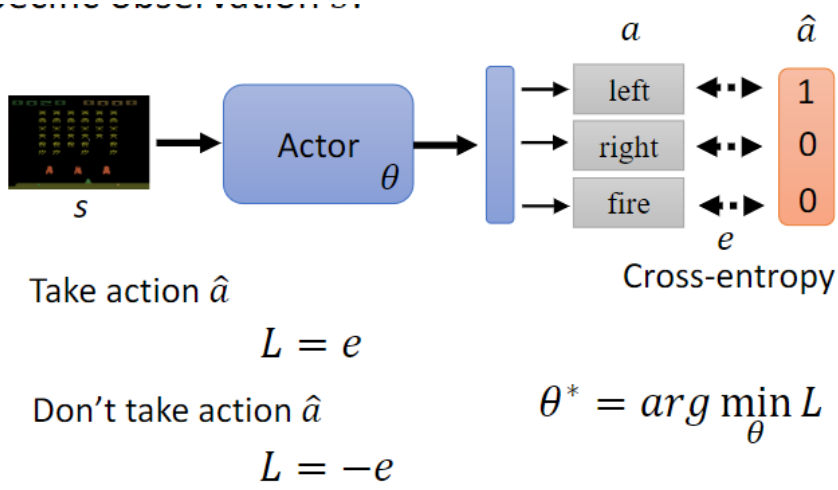
Policy Gradient

> 拿来解RL做optimization的一个常用的演算法

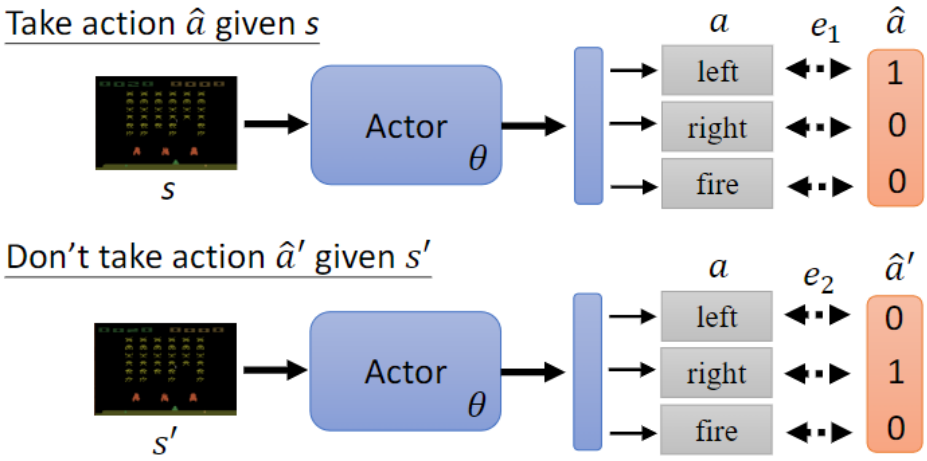
如何操控一个actor的输出

目标：当给定一个特定的observation s 时就输出或不输出一个给定的action $\hat{a}$。

可行的一种方法：做一个supervised learning，将收集的$\hat{a}$作为label（ground truth），设计loss函数计算a和$\hat{a}$之间的交叉熵，当作一般的监督学习处理（如下图）



给定s期望采取$\hat{a}$，给定s'期望不要采取$\hat{a}'$——对$(a, \hat{a})$交叉熵尽量小，对$(a, \hat{a}')$交叉熵尽量大。设计loss函数$L = e_1 - e_2$

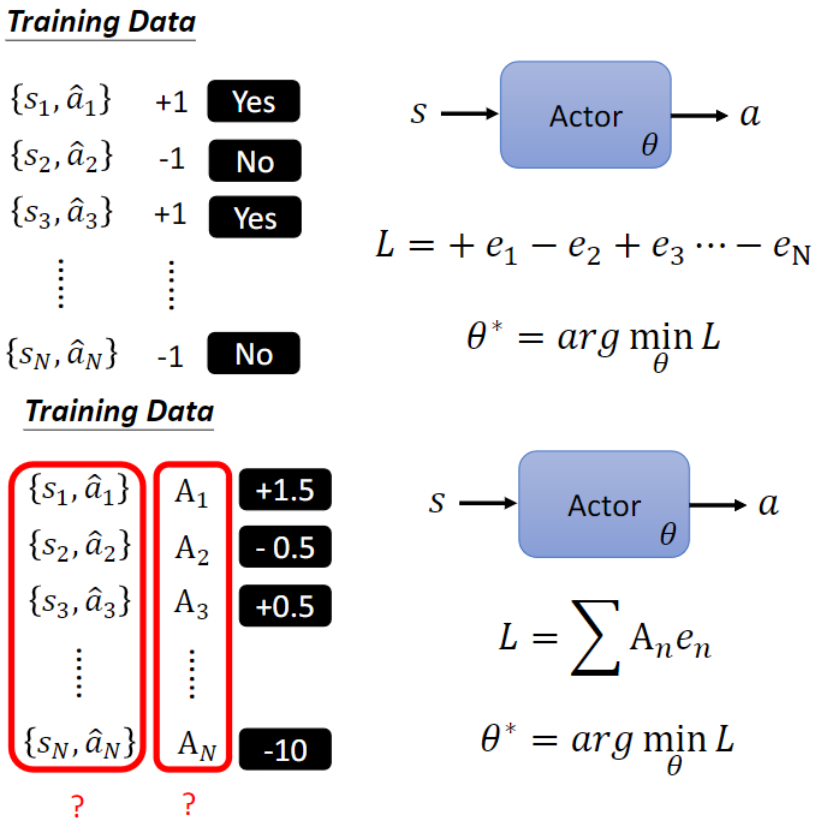


$$L = e_1 - e_2 \qquad \theta^* = arg \min_{\theta} L$$

藉由监督学习的做法来操纵actor的输出。期望actor不采取$\hat{a}'$意味着期望actor采取除此之外的其他action（不一定是什么都不做）。在某一个环境下，我们也可以让某个actor既期望做某个action，又不期望做另外一个action（或许会出现矛盾的情形？）。

以上部分，比较重要的部分就是收集资料/标签。从最简单做和不做（binary problem）处理，事实上对数据标签更多的是范围模糊化，即$\{-1,1\} \rightarrow [-k,k]$，给期望量化出一个分值。处理方式的改变对损失函数的设计也稍有不同。（如下图）

$$L = +e_1 - e_2 + e_3 - \cdots + e_N \rightarrow L = \sum A_n e_n \tag{2}$$

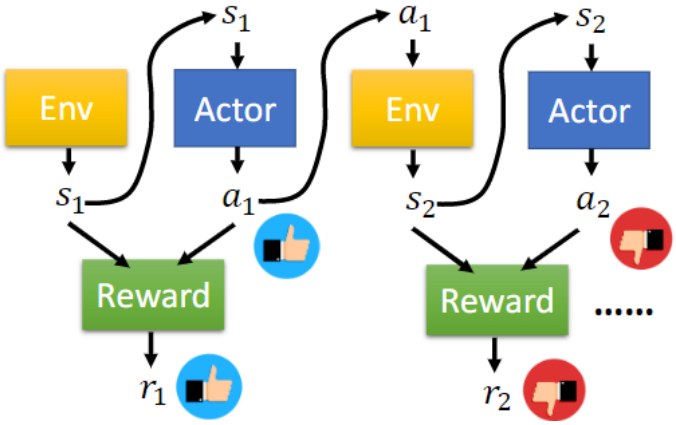


后面的处理是一样的，train出这个θ^*。而难点在于如何定红框中的$A_i, i-1,2,\ldots,N$；以及我们该如何产生$(s_i, \hat{a}_i)$这样的pair呢?

上右图中通常是我们制作数据集的一种方式，$e_i = corss\ entropy(s_i, \hat{a}_1)$，整个流程和监督学习相差无二，以下陈述如何确定Action的分数（权值）A，不同的RL方法通常就是在A上面做文章，完全可以以问题为导向，大胆假设，对A进行合理的定义。

如何得到A？ Version 0：*immediate reward*

- 首先需要收集一些训练资料：pair$(s_i, \hat{a}_i)$。我们用actor去跟环境做互动，记录actor产生的s和a。这个actor我们可以认为是随机的



我们把输入s给actor所得到的a都统统记录下来。通常我们会做多个episode，以收集足够多的资料。我们再评估这些记录，来标记这些结果是好还是不好，标记为$A_i, i = 1, 2, \ldots, N$

Training Data

$$\begin{matrix} \{s_1, a_1\} & \mathrm{A}_1 = r_1 \\ \{s_2, a_2\} & \mathrm{A}_2 = r_2 \\ \{s_3, a_3\} & \mathrm{A}_3 = r_3 \\ \vdots & \vdots \\ \{s_N, a_N\} & \mathrm{A}_N = r_N \end{matrix}$$

最简单的评估的方法：假设在某个s输入给actor，他采取了记录中对应的action a，假如reward是正向的（positive），那么说明这个action可能更倾向于是好的。 因此可以认为A的取值和reward的输出是成正比的，所以直接让

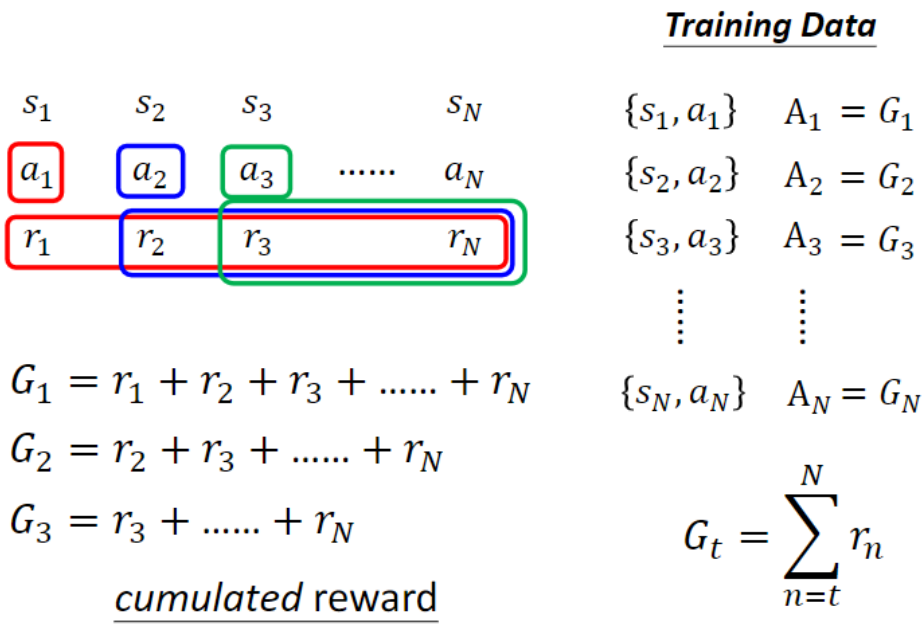
$$A_i = r_i, \ i = 1, 2, \ldots, N \tag{3}$$

这个version 0是一个短视近利的actor（Short-sighted Version），完全没有长远规划。

- 局限性：
  - 一个action通常会影响到接下来的observation，再影响到之后的reward.....每个action并不是独立的，影响会传递下去；这一点version 0没有考虑到。
  - *Reward Delay*：正如珍珑棋局虚竹的神之一手，要想获得长远的利益（long-term reward）甚至有必要牺牲眼前蝇头小利（immediate reward）。以上述Atari游戏为例，为了击杀alien，同时需要飞船不断的左右移动调整。
  - 在version 0中，space invader游戏中，正向的action只有开火，而左右移动都不被鼓励。最后的结果就是学会的玩家只会无脑开火。

如何得到A？ Version 1：*cumulated reward*

- 对于所有observation $\{s_1, s_2, s_3, \ldots, s_N\}$，针对各自action $\{a_1, a_2, a_3, \ldots, a_N\}$，只考虑其序列之后的所有情况。该方法称之为*cumulated reward*



如上图所示，~~（感觉有点马尔可夫的影子了hhh）~~

$$A_i = G_i = \sum_{n=i}^{N} r_n,\ i = 1, 2, \ldots, N \tag{4}$$

- version 1的结果可以让左右移动（sapce invader游戏）也有正向的鼓励。
- 局限性：考虑$A_1 = G_1 = r_1 + r_2 + \cdots + r_N$，对于action a_1真的会的最后的r_N的结果产生重要的影响吗？简而言之，version 1缺点在于“管的太宽了”，一刀切让每个action都对后面所有reward结果产生影响。

如何得到A？ Version 2：*discounted cumulated reward*

- 针对以上这个局限性，做一个改进：设定一个discount factor $\gamma < 1$，让 $A_1 = G_1' = r_1 + \gamma r_2 + \gamma^2 r_3 + \cdots + \gamma^{N-1} r_N$，通常会打个九折或九九折。如此一来，$a_1$对$r_N$的影响力就微乎其微了。

$$A_i = G_i' = \sum_{n=i}^{N} \gamma^{n-i} r_n,\ i = 1, 2, \ldots, N \tag{5}$$

> 形如$\{s, a\}$是一个step，或者说是一笔资料。一个episode是由许多个observation和action共同组成的。

如何得到A？ Version 3：*discounted cumulated reward with normalization*

> 好和坏、Reward是相对的，我们需要对分数设计做一个标准化。

有些游戏中只能拿到大于等于0的分数，这会导致误判一些action是好的，这就需要标准化reward。以下介绍几种方法：

- 最简单的一种方法：每个分数标记/权重减去一个常数b，b在文献中通常称之为"Baseline"。

$$\begin{aligned}
\{s_1, a_1\} \quad & A_1 = G_1' - b \\
\{s_2, a_2\} \quad & A_2 = G_2' - b \\
\{s_3, a_3\} \quad & A_3 = G_3' - b \\
\vdots \quad & \vdots \\
\{s_N, a_N\} \quad & A_N = G_N' - b
\end{aligned}$$

让一些分数较高（偏正面的）的值为正，让一些偏负面的值为负。具体如何设定这个baseline之后会讲到。

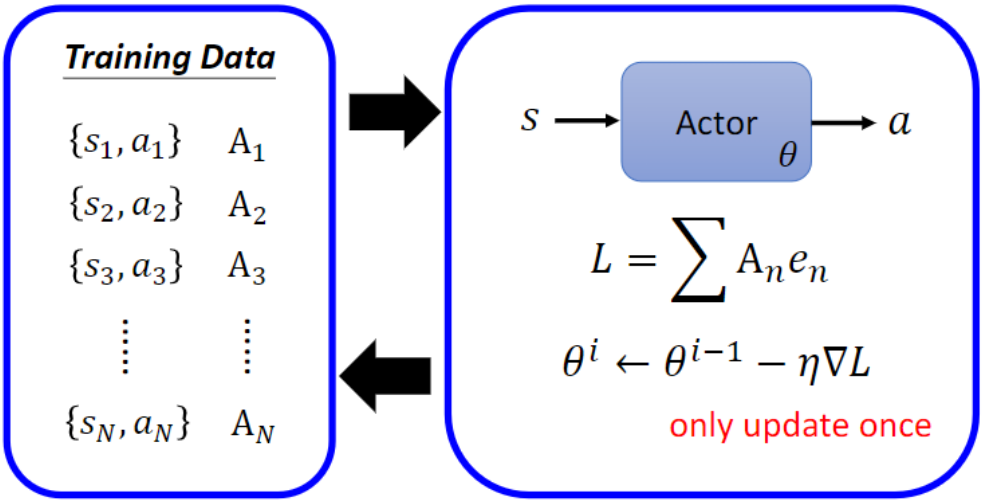
> 有提到可以heuristic function（启发式学习），国际象棋的程式deepblue就用到了这个，中间过程会实时的输出reward

Policy Gradient是怎么操作的?

步骤描述如下：

- 随机初始化Actor（通常是NN）的参数θ^0
- 开始train，整个training iteration $i = 1 \rightarrow T$：
  - 使用actor θ^{i-1}去跟环境做互动
  - 收集到数据对$\{s_1, a_1\}, \{s_2, a_2\}, \ldots, \{s_N, a_N\}$
  - 对数据进行评价，计算$A_1, A_2, \ldots, A_N$——最关键的一步，上述给出了4中计算reward的方式
  - 解优化问题（最大化reward），计算损失函数L
  - $\theta^{i-1} - \eta\nabla L \rightarrow \theta^i$，其中$\eta$是学习率（learning rate），梯度下降

和一般的deep learning不同的是，DL中训练资料或测试资料相对于迭代过程通常是静态的；而Policy Gradient的过程表明在RL中收集资料在迭代过程中完成；迭代多少次，就要收集资料多少次。事实上在RL中，每个循环（loop）中梯度下降是只能更新一次（如下图所示）；更新完一次参数，又得下去重新收集一次数据。



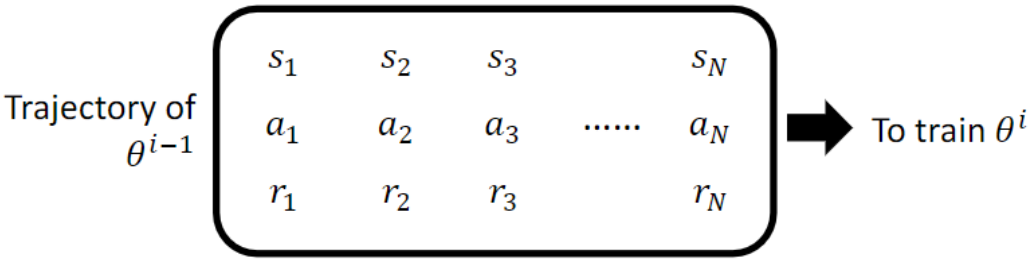
每次更新一次模型参数。都需要重新收集整个训练数据。θ^{i-1}的决策（actions）不一定对θ^i有用。

原因在于每次数据都是actor（某个状态时，每个状态都是一个actor θ）和环境交互得来的，不同状态的actor的数据对很可能是完全相反的（譬如说对于同一个action的评价...），总而言之，one's drug could be one's poison，同一个action对于不同的actor而言，它的评价是不一样的.（这里老师举了"棋魂"的例子，orz我没康过）

所以说在RL做梯度下降非常费时间。

On-policy v.s. Off-policy

- ***On-policy***：训练的actor和交互的actor是同一个
- ***Off-policy***：训练的actor和交互的actor可以不是同一个
  - 在off-policy的learning中显而易见的好处：可以想办法拿θ^{i-1}的actor的策略来训练θ^i的actor



在这个方法下，我们不用每次update梯度后就重新收集全面数据。

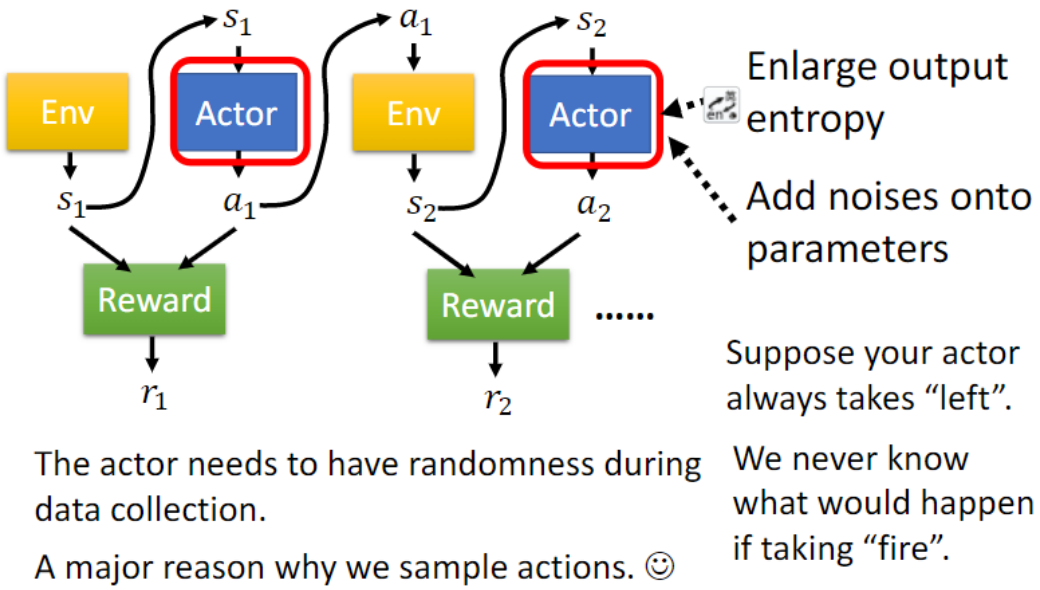
- 非常经典的一种off-policy的做法：*Proximal Policy Optimization (PPO)*（~~貌似还挺有趣的~~ 😁，~~之后自行了解叭~~）
- **off-policy的关键**在于*train的actor*要知道自己和*和环境交互的actor*之间的差异

> 怪怪的例子：interacting actor：Chris Evans（米国队长）；training actor：Me
> 如果我和Chris要追女孩，他的策略和我的策略显然是有差异的。（人帅真草，人丑吃草？？）

Exporation

> tips：尽量让模型去收集更为丰富的训练资料

如上所述，actor在采取action的时候是具有一定随机性的，这个随机性非常重要，随机性不够可能会train不起来——以space invader为例，如果没有足够随机性选择action（开火），那么可能整个训练过程就是在左右移动或不动，从头到尾都没有actor将action考虑在内。只有随机性足够，一个actor选择去吃螃蟹，才会有之后的评估，从而使得收集资料更加丰富。



为了让actor随机性大一些，在training的时候我们可以刻意的加大一些随机性，例如说actor的输出是一个distribution，可以通过加大distribution的entropy或在actor的参数上加noise来增大其选择action的随机性，让其训练时比较容易sample到几率比较低的action。

exploration就是RL中比较常见的一个trick，让actor去尝试尽量不同的action，否则很有可能会train不出好的结果。

Actor-Critic

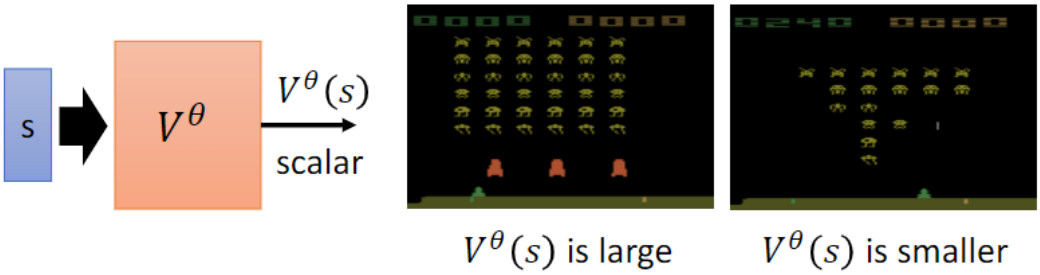
以上讲述的RL都是去learn一个actor，而这部分就是讲述RL如何去learn一个critic

什么是Critic？

所谓critic：给定一个参数组θ，评估这个actor看到某个observation s（采取对应的action a）后它能得到多少的reward

critic实例介绍 ***Value Function*** $V^\theta(s)$：当参数组为θ的actor看到observation s 后，接下来它得到的*discounted cumulated reward*（见上version 3）：$G'_1 = r_1 + \gamma r_2 + \gamma^2 r_3 + \dots$

通常G'得玩完整场游戏才会知道，Value Function要做的事情就是在中间过程预测这个G'



函数的输入为 s ，上标 θ 表明其观察的对象为参数组为θ的actor，V^θ是函数，输出一个标量（scalar）。

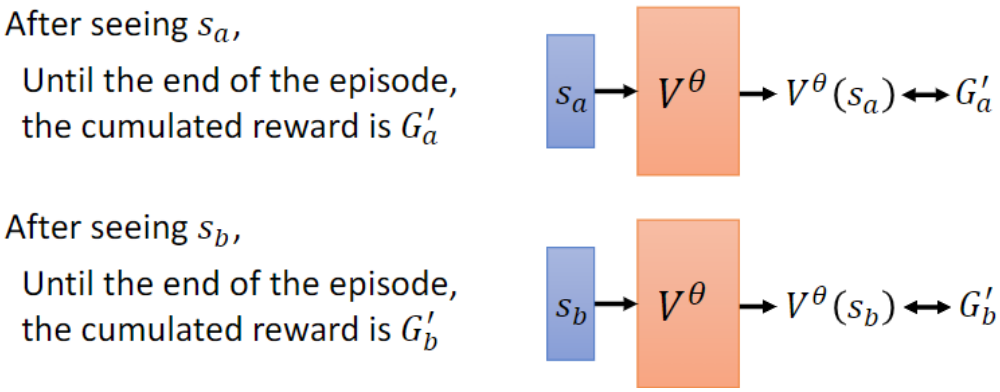
输出结果的解释：当画面中alien数量较多，value function对G'得分期望较大，而alien数量较少时（游戏的末尾段），此时对reward的期望较小。除此之外，注意到value function的结果时跟观察的对象（actor）有关系的，再以该游戏为例，如果actor比较菜（alien多的时候actor很容易被杀死），此时可能$V^\theta(s)$结果比较低。

Critic是如何被训练出来的?

> 两种常见的训练方法：第一种是马尔可夫蒙特卡洛方法（MC）；另一种是时序差分算法（TD）。前者比较符合直觉

Monte-Carlo（MC） based approach

Actor和环境做互动，会得到游戏的一些记录。 当actor看到（sample到）observation s_a 时，接下来的*discounted cumulated reward*就期望是G'_a。如下图所示，在MC机制下，当函数V^θ给出输入为s_a时，其输出$V^\theta(s_a)$应该要和G'_a越接近越好。



MC方法非常符合直觉，我们通过了解actor决策并得到reward（玩完整场游戏）从而获取一笔训练资料，直接拿这些训练资料来train我们的critic V^θ。

Temporal-difference (TD) approach

> 不用玩完整场游戏就可以得到训练Value Function的资料

TD机制下，我们要获得

$$\dots s_t, a_t, r_t, s_{t+1}, \dots \tag{6}$$

这样一个序列。解释：actor看到s_t，执行a_t，获得reward记作r_t

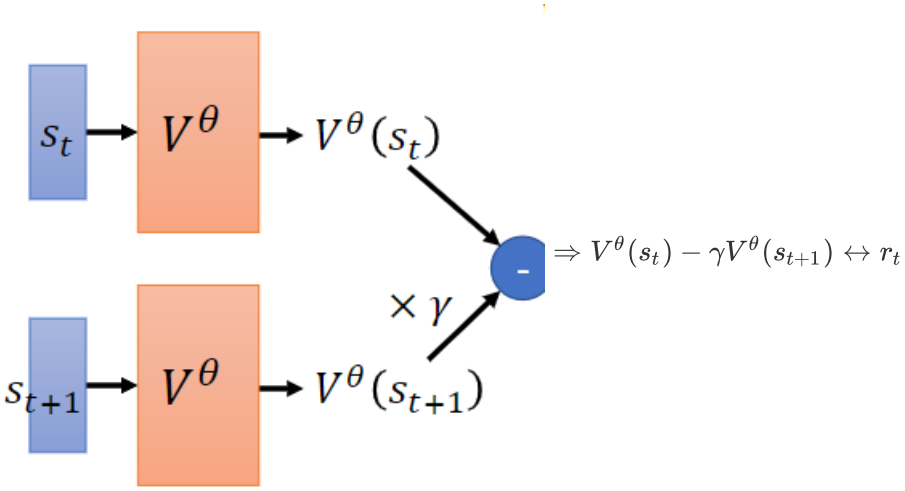
只需要看到这样一个序列（部分），我们就能拿来更新$V^{\pi}(s)$的参数了（train价值函数）。这个方法的好处在于：与MC方法相对比，后者需要做完整场游戏方能获得一笔训练资料；事实上，一些现实任务中（游戏）可能持续时间比较长（甚至可能永不结束），在这个情景下，我们更倾向于使用TD的方法。

以下详细说明TD方法如何在部分序列情况下训练V^{θ}函数（更新参数）

将看到s_t之后的reward记作$V^{\theta}(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2}...$以此类推
$V^{\theta}(s_{t+1}) = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3}...$显然可以得到$V^{\theta}(s_{t+1})$和$V^{\theta}(s_t)$的关系：

$$V^{\theta}(s_t) = \gamma V^{\theta}(s_{t+1}) + r_t \tag{7}$$

当我们获得公式（6）这样一串序列，train的时候使其尽量满足公式（7），如下所示



换言之，已知一笔资料r_t，同时知道s_t, s_{t+1}，我们train的目标就是使得$V^{\theta}(s_t) - \gamma V^{\theta}(s_{t+1})$和$r_t$越接近越好。这就是TD方法，用来train出价值函数的未知参数θ

MC v.s. TD

举个例子：某个actor和环境互动了八个episode，游戏比较简单，通常只持续1到2个回合

The critic has observed the following 8 episodes

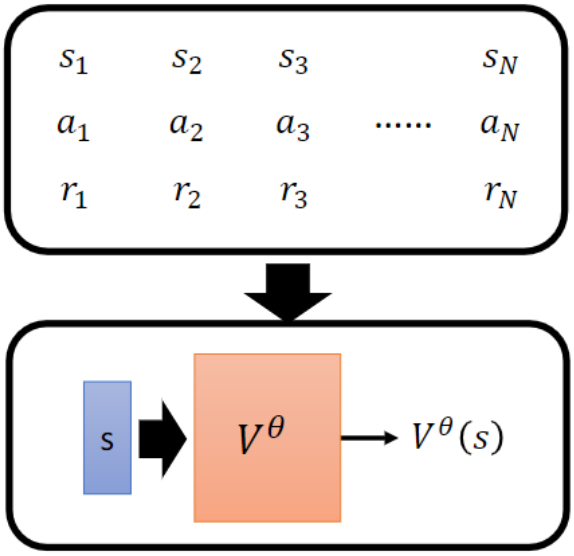
- $s_a, r = 0, s_b, r = 0$, END
- $s_b, r = 1$, END
- $s_b, r = 1$, END
- $s_b, r = 1$, END
- $s_b, r = 1$, END
- $s_b, r = 1$, END
- $s_b, r = 1$, END
- $s_b, r = 0$, END

这里假设无视action，然后设置$\gamma = 1$（不做discount），$V^{\theta}(s_b) = \frac{3}{4}$，而$V^{\theta}(s_a) = 0$或者$\frac{3}{4}$——前一个答案是按MC方法的得到的，后者则是TD方法算出来的。MC方法给出的前提假设就是s_a, s_b两者有联系，而在TD中两者无关联。

Critic 如何应用在action的训练当中?

version 3.5

上文已经给出了如何得到评估A的version 3.



Training Data

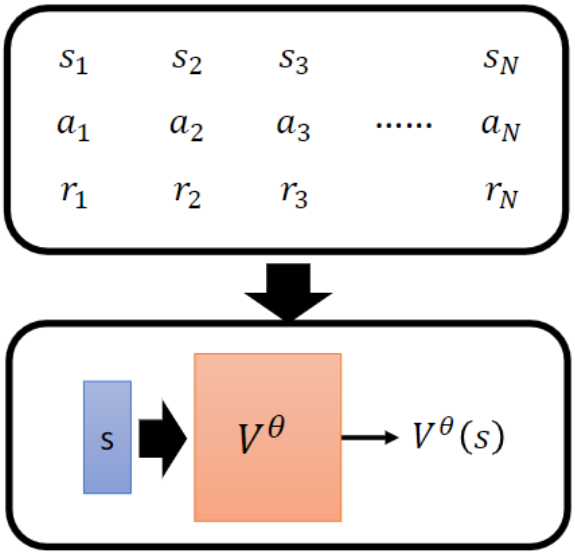
$\{s_1, a_1\}$ $\mathrm{A}_1 = G'_1 - b$

$\{s_2, a_2\}$ $\mathrm{A}_2 = G'_2 - b$

$\{s_3, a_3\}$ $\mathrm{A}_3 = G'_3 - b$

$\{s_N, a_N\}$ $\mathrm{A}_N = G'_N - b$

在critic中，把state s打入V^θ中输出$V^\theta(s)$，我们把$V^\theta(s)$当作b，得



Training Data

$\{s_1, a_1\}$ $\mathrm{A}_1 = G'_1 - V^\theta(s_1)$

$\{s_2, a_2\}$ $\mathrm{A}_2 = G'_2 - V^\theta(s_2)$

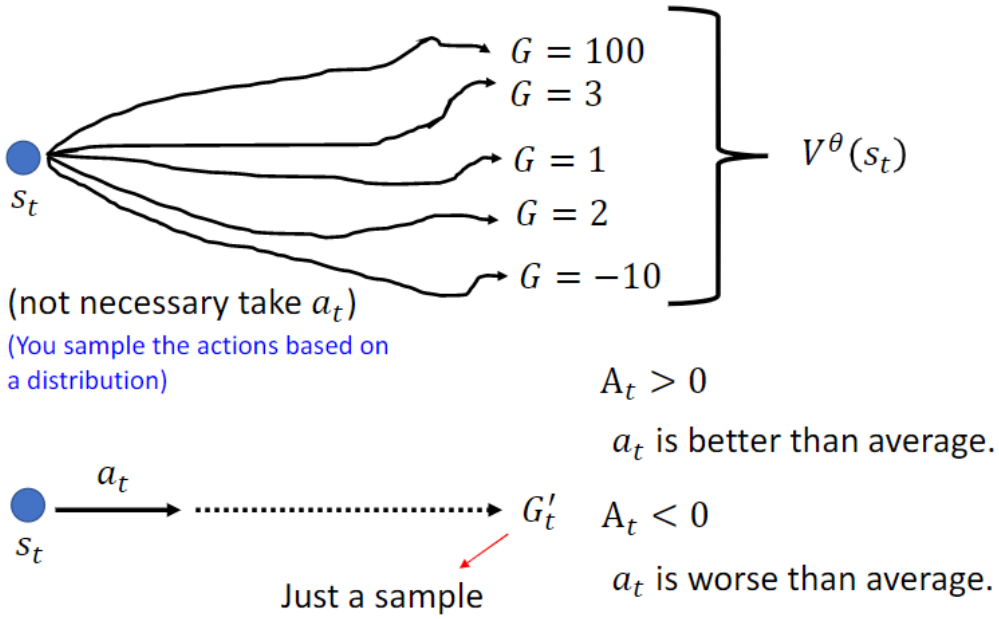
$\{s_3, a_3\}$ $\mathrm{A}_3 = G'_3 - V^\theta(s_3)$

$\{s_N, a_N\}$ $\mathrm{A}_N = G'_N - V^\theta(s_N)$

根据左边的训练资料可以learn出一个critic，然后这个critic产生出一个state，输入到价值函数中，然后利用输出值（分数）进行标准化。针对数据对$\{s_t, a_t\}$，我们有

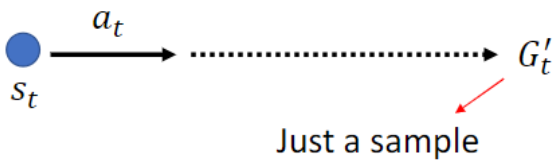
$$A_t = G'_t - V^\theta(s_t) \tag{8}$$

来定义对训练数据中对决策的评估。



当看到s_t时，不一定会执行a_t这个action，actor的输出实际上是一个的distribution（action的概率空间上，每个action都会有几率，不断sample空间里的action）因此，对于同一个s_t会有不同的G（cumulative reward without discount），把这些可能的结果平均起来就是$V^\theta(s_t)$。

所谓G'_t，即在s_t前提下执行a_t后一路玩下去最后得到一个reward就是G'_t

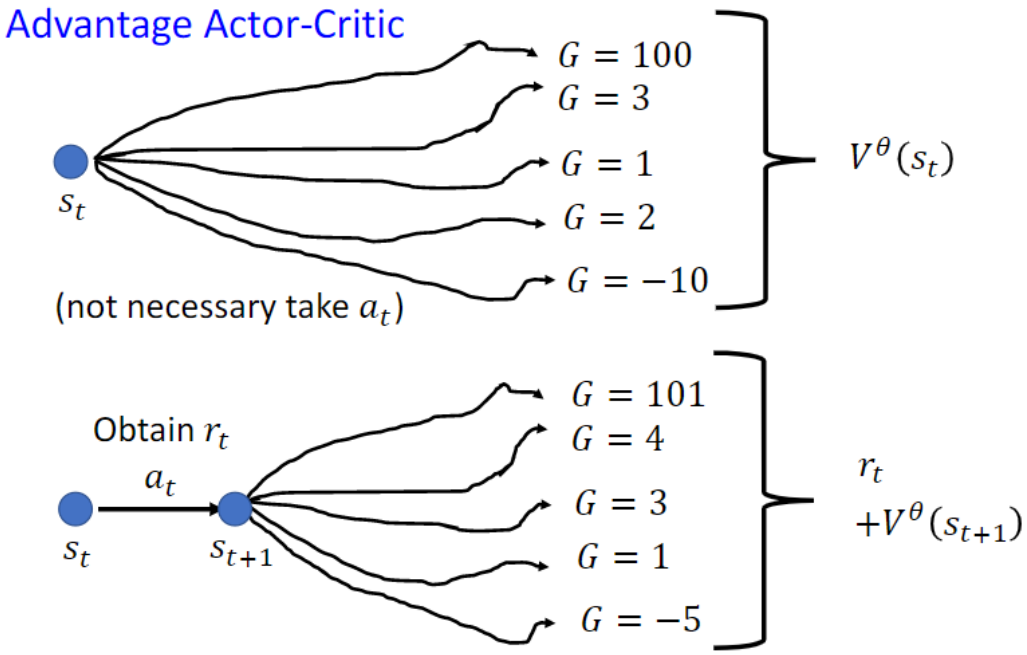


如果$A_t > 0$，说明$G'_t > V^\theta(s_t)$，这时候说明上图中的a_t是要比上上图中random sample到的所有a（action）都要好（比平均水准好）；反之，若$A_t < 0$，则说明$G'_t < V^\theta(s_t)$，即这时候a_t在游戏中属于低于平均水平的下策。以上简单直觉的解释了为什么要按公式(8)来计算A。

这里的version 3.5是拿一个sample出来的action下的reward减去按distribution去random sample出来action下的平均的reward。以下陈述version 4，与用sample减去平均的不同，其用平均的减去平均的。

version 4：Advantage Actor-Critic

如下图所示，上部分内容上文解释了，以下解释下部分内容：



当看到s_t，执行完a_t后就跑到下一个场景s_{t+1}，然后在这个场景下，去构成一个action的distribution，类似的，把cumulative reward平均起来得到期望的reward为$V^\theta(s_{t+1})$，那么从s_t出发总的期望的reward为$r_t + V^\theta(s_{t+1})$

因此，针对数据对$\{s_t, a_t\}$，我们有

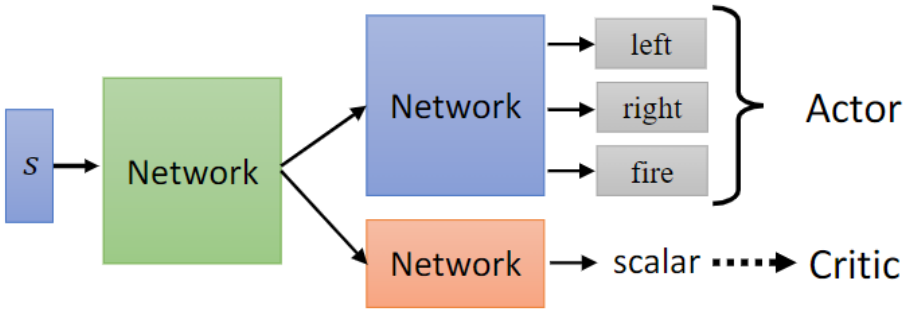
$$A_t = r_t + V^\theta(s_{t+1}) - V^\theta(s_t) \tag{9}$$

这个过程很像是对照实验，以减号为界分成两项，前一项时看到s_t去采取a_t后再去sample一些action的reward总和，后一项则是看到s_t没有采取a_t这个action就直接去sample一些action的期望reward，两者之差作为A_t来评估action a_t的好坏。显然地，A_t越大说明这个action a 越好。（这方法很符合直觉啊。。owo）

Actor-Critic的小tip

> tricks for homework ^ o ^：让actor和critic共用一部分网络/参数

Actor是一个network，输出为游戏的画面，输出每一个action的分数（类似分类器）；critic也是一个network，输出是一个数值（scalar），代表接下来得到的cumulative reward。如下图，这两个network有着同样的输入，它们应该有部分的参数是可以共用的。俩网络都需要理解输入的游戏画面s，所以其共用前面几个layer（前部的network）



other notices

在RL中sample的好不好（非常重要）关系到最后能不能train好，通常sample的越丰富越好。
critic中的V^θ就相当于一般生（average grades），超过它这个action就是好，低于它就是捞。

关于actor的distribution: actor这个网络的输出类似于一个分类器（每个action对应一个值），那对这个结果做一个normalization，每个action就对应一个几率，critic计算A的actor就按照这个几率的distribution来sample一些action

*outlook: Deep Q Network（DQN）

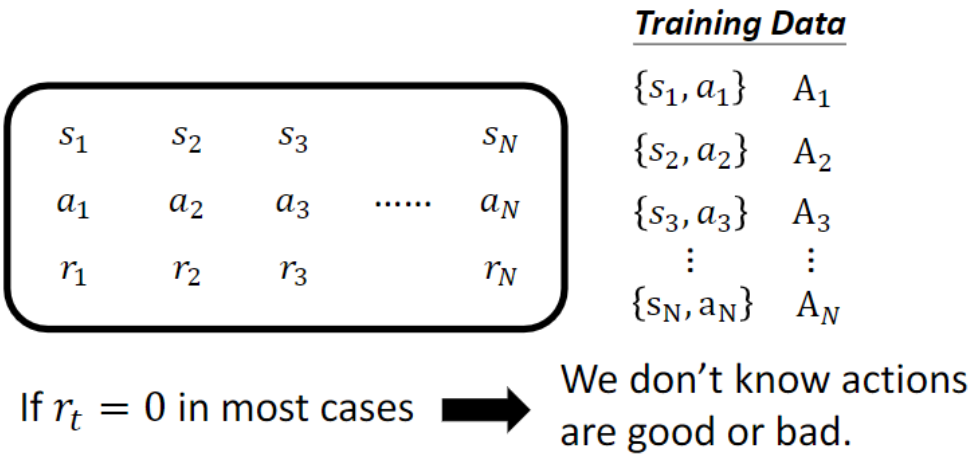
蛮犀利的一种做法：直接用critic来决定采取哪个action。感兴趣自行了解。DQN有非常多的变形，其中比较知名的是rainbow

~~【学习DQN，此处待拔草owo】~~

Reward Shaping

> When reward is sparse

目前为止，我们理解的RL就是把actor（agent）拿去和环境互动，然后得到一堆reward，对reward做一些处理（version 1~4）得到这边的分数A（见公式9），用A来教actor该做些什么（actions）。但是，在一些实际的情景中，多数的时候reward都是0，只有非常低的几率得到非0的正向的reward，这种现象称之为***Sparse Reward***。此时，在以上所设计的评估体制下，training data的大部分几乎每个action的评估都是差不多的（不知道是好是坏），这种情况下很难去train好我们的actor。（e.g. 教机械臂拧螺丝，以上述思路设计reward，大概率根本train不出来，小概率是巧合）



Sparse Reward的现象和对弈很像，整个过程中几乎不会得到正向或是反向的reward，只有整场游戏结束才会知道reward（win or lose）。但下围棋至少局终未一定会得到一个有用的reward。

面对Sparse Reward的问题，通常我们会设计**额外的reward**来引导actor（agent）学习——即在真正要去optimize的reward之外去定义一些额外的reward来帮助agent学习，这种方法称之为***Reward Shaping***

Reward Shaping 实例：在Vsidoom游戏的情景下

> Visdoom：FPS游戏，被当作某AI比赛的场景（Visual Doom AI Competition @ CIG 2016），rk1的文章https://openreview.net/forum?id=Hk3mPK5gg¬eId=Hk3mPK5gg

这篇文章用了reward shaping的概念。

Parameters	Description	FlatMap | CIGTrack1
living	Penalize agent who just lives	-0.008 / action
health_loss	Penalize health decrement	-0.05 / unit
ammo_loss	Penalize ammunition decrement	-0.04 / unit
health_pickup	Reward for medkit pickup	0.04 / unit
ammo_pickup	Reward for ammunition pickup	0.15 / unit
dist_penalty	Penalize the agent when it stays	-0.03 / action
dist_reward	Reward the agent when it moves	9e-5 / unit distance

以上这些parameters并不直接增益游戏分数，属于是人类教它如何积极的卷，摆烂就扣分。

*Reward Shaping无疑是需要人类对环境的理解来做额外的这些rewards（实际就是domain knowledge）具体问题具体分析。

Reward shaping - Curiosity

> curiosity based reward shaping. 来自https://arxiv.org/abs/1705.05363

这种方法假设给machine加上“好奇心”，当actor看到**new** but **meaningful** thing时就得到积极的reward

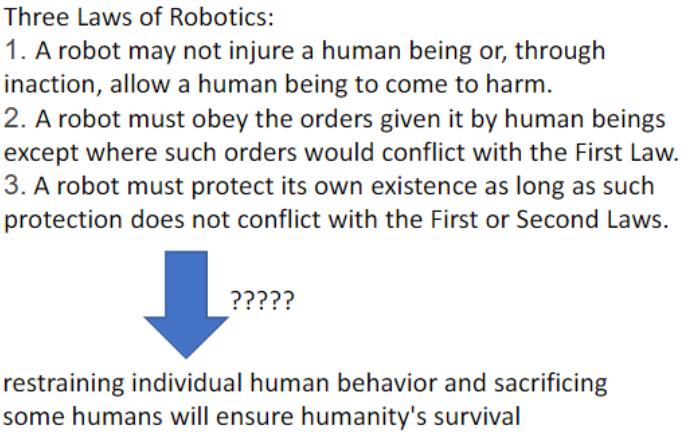
No Reward: Learning from Demonstration

> 如果reward都没有，该怎么去做RL

Motivation

- 通常只在一些artificial的环境里边（譬如游戏）可以能容易准确定义reward；而在一些实际任务中，reward非常难被define（譬如RL做自动驾驶）
- 在一些不好定义reward的场景中，我们（人类）可以自己想一些reward出来。Hand-crafted rewards can lead to uncontrolled behavior（e.g. Reward Shaping）

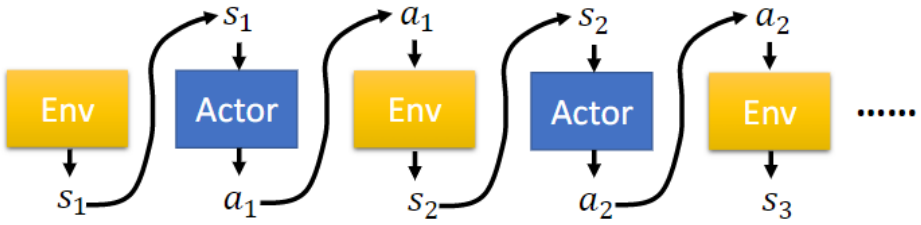
  这种机制不好的例子👇（阿西莫夫三大定律-->“机械公敌”出现的场景：为了保护人类-->把人类圈禁起来？？）



  有时候，人为设计的rewar不一定时最好的。

Imitation Learning

在没有reward的情况下，训练actor/agent来和环境做互动——Imitation Learning

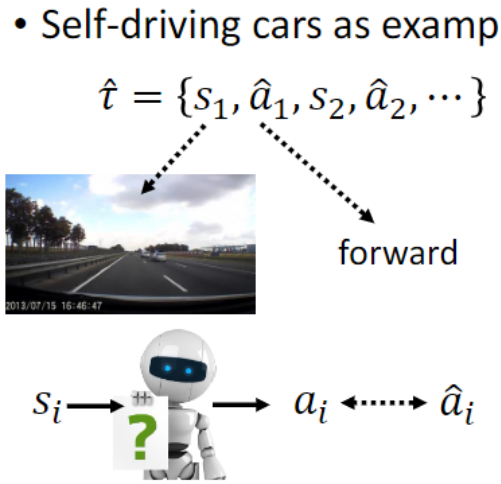


在IL里面，假设actor仍然会和环境做互动，不过没有reward。但是我们可以建立“expert的示范”，例如说我们可以找很多人类，让人类也跟环境做互动，把这些记录下来就叫做expert 示范，将其记作 τ。actor就可以根据 τ 来进行学习。

我们有$\{\hat{\tau}_1, \hat{\tau}_2, \ldots, \hat{\tau}_K\}$，其中每个 $\hat{\tau}$ 都是expert的trajectory，譬如说：针对自动驾驶有人类司机的驾驶记录、train机械臂前人类拉着机械臂手动规划路径

实例

IL非常像supervised learning，以自动驾驶为例：
有人类驾驶员记录作为expert示范：$\hat{\tau} = \{s_1, \hat{a}_1, s_2, \hat{a}_2, \dots\}$，$s$就是实时道路场景，$\hat{a}$就是人类司机面向场景采取的行为。



最简单直接的想法：让machine去模仿人类的行为，machine给出的action为a_i越接近人类行为$\hat{a}_i$越好，这种做法称之为*Behavior Cloning*。

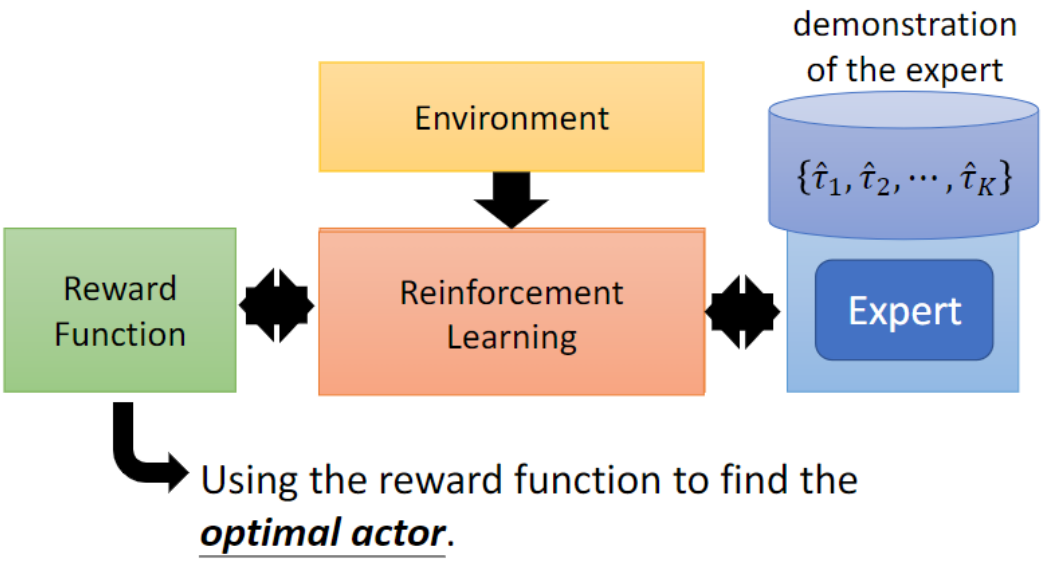
这种只让machine去完全模仿人类的行为的做法会有几个问题：

- machine和人类所观察的observation会是不一样的，人类在驾驶任务上会大概率的避免许多问题（譬如很多事故现场），expert仅仅只能sample出非常有限的observation，而对machine小白而言，它预期会遇到的情况（observation）在training data中根本不会存在。
- expert的一些行为是多余的（通常只是expert与任务无关的行为），而machine亦步亦趋的完全模仿expert并不合理。machine并不知道什么是需要学到，什么是不需要。假设machine的学习能力有限的前提下，在任务过程中也只能选择部分行为去学习，这些多余动作"噪声"的存在让Behavior Learning造成很大的困扰。

为了解决第二个问题，提出了Inverse Reinforcement Learning

Inverse Reinforcement Learning

> 正如字面意思，和一般RL任务背驰，IRL根据expert示范和环境来反推出reward应该长什么样子。换言之，让machine自己来定义reward，reward function可以被学出来。然后再用这个learn出来的reward来做一般的RL任务。

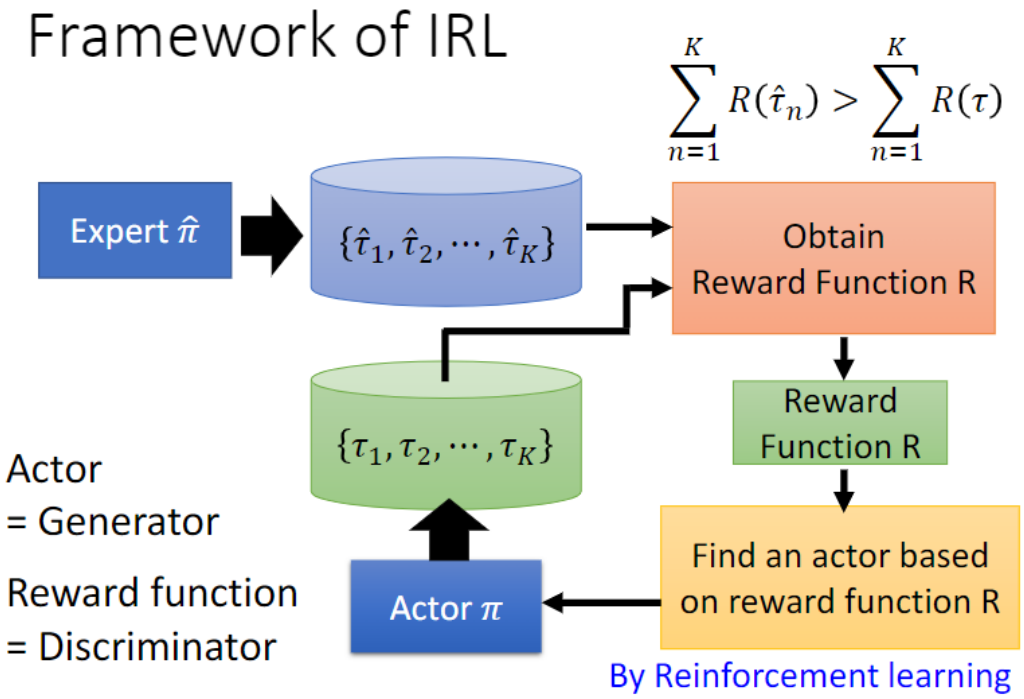


我们可以通过reward function来learn到optimal actor。简单的reward function不一定会learn出简单的actor

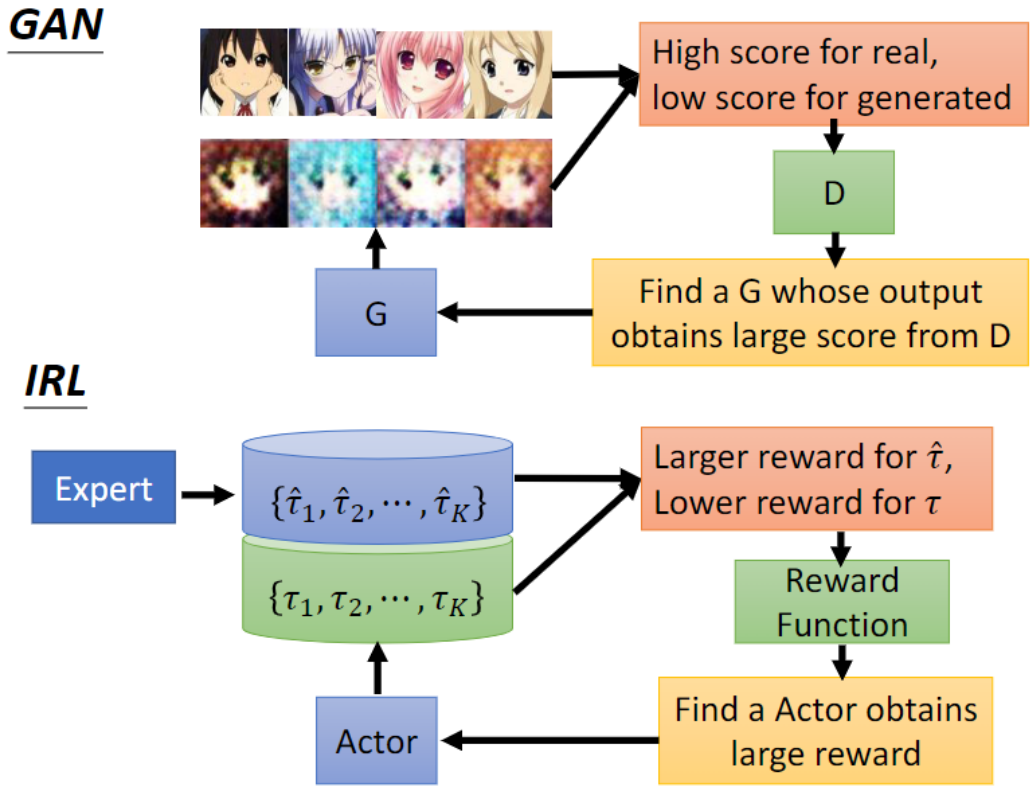
Details :

- 原则：老师（teacher）的行为总是最好的（best）——并不代表完全要模仿teacher的行为，只是表示teacher这个actor的action都能取得最高的reward。teacher就是expert的示范。
- 基本思想：
  - 初始化一个actor
  - 迭代：每一轮
    - 一个新的actor和environments做交互，以获得相对应的trajectories
    - 定义reward funtion，要求在这个reward定义下，老师的trajectories要给较高的分数，而actor要给较低的分数。
    - update actor的参数：基于上步骤给出的reward funtion，去maximizes这个actor的reward（更新actor网络的参数）
    - 循环上述步骤
  - 输出learn出来的reward funtion和actor

这个流程很像GAN: Actor：“生成器”；Reward function：“判别器”



IRL框架流程和GAN的对比，有异曲同工之妙



IRL技术通常会应用在训练机械臂。人类手把手教machine如何做对应任务，然后通过IRL技术train机械臂来学会任务。

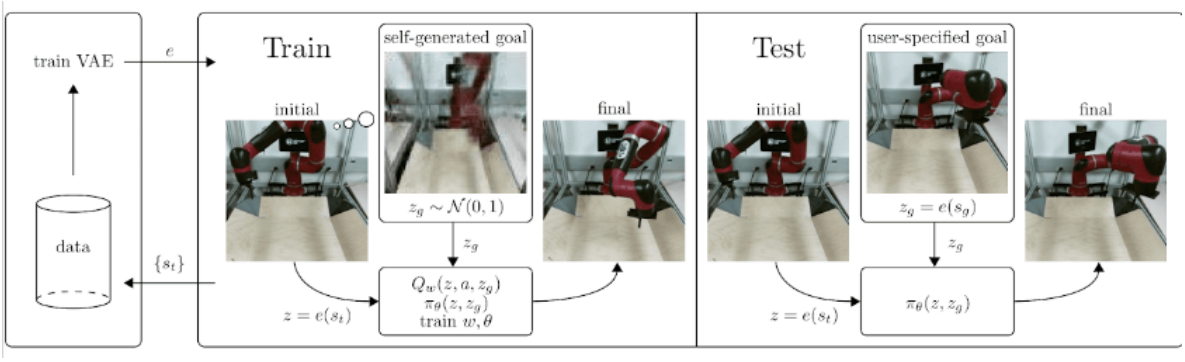
这类方法一般被称之为基于demonstration的RL方法（Learning from Demonstration）。

Outlook

与IRL不同，不手把手教而是给机器一个画面，让machine做出画面上的行为。——Reinforcement learning with Imagined Goals (RIG)

可阅读的参考文献如下

- Visual Reinforcement Learning with Imagined Goals, NIPS 2018
- Skew-Fit: State-Covering Self-Supervised Reinforcement Learning, ICML 2020



train过程很有意思，让machine自己创造一些目标（self-generated goal），然后达到目标...

题外话：IRL并不一定需要完全模仿人类的行为，也可以在模仿行为之外增加一些额外的reward，或许会让machine在任务上做的比人类更加出色。