**强化学习和监督学习对比：**

1.监督学习，对有标注的数据进行学习。有分类和回归两类。以分类为例，训练过程中，我们给神经网络真实的标签，比如现在输入了汽车的图片，它预测出来是飞机。我们就会直接告诉它，你这个预测是错误的，正确的标签应该是车，并通过损失函数进行反向传播来训练网络。

监督学习中两个假设：  
 （1）数据有标签

（2）数据没有关联

有标签很好理解，我们得知道答案才能进行指导。没有关联就是我们希望输入的数据具有总体代表性，如果数据不具有总体代表性，那么就是特例的情况，那规律就会总结得不好或是错误，因为这些规律是由个例推算的，不具有推广的效果。比如训练矩阵分类器，输入数据都是有关联的，例如正方形，那么学习到的模型就只能判别正方形是矩形，因为他不知道长宽不等的长方形也是矩形。数据应该要满足独立同分布，独立代表数据之间无关联，同分布说明数据样本服从同一个分布，比如掷骰子，就服从同一个分布，每个数字的概率都是1/6。

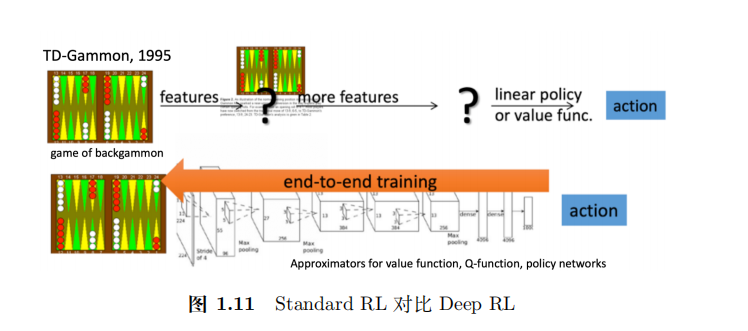
1. 强化学习数据是相关的时间序列数据，不满足独立同分布，一段时间内智能体采取的动作是连续性，有关联。在强化学习里面，环境可能会告诉你这个行为是错误的，但是它并没有告诉你正确的行为是什么。而且更困难的是，它可能是在一两分钟过后告诉你错误，它再告诉你之前的行为到底行不行，这就是延迟奖励 (Delayed Reward)。

现实生活中强化学习的例子：

在自然界中，羚羊其实也是在做一个强化学习，它刚刚出生的时候，可能都不知道怎么站立，然后它通过试错的一个尝试，三十分钟过后，它就可以跑到每小时 36 公里，很快地适应了这个环境。

**强化学习和深度强化学习：**

1. Standard RL：之前的强化学习，比如 TD-Gammon 玩 backgammon 这个游戏，它其实是设计特征，然后通过训练价值函数的一个过程，就是它先设计了很多手工的特征，这个**手工特征可以描述现在整个状态**。得到这些特征过后，它就可以通过训练一个分类网络或者分别训练一个价值估计函数来做出决策。
2. Deep RL：现在我们有了深度学习，有了神经网络，那么大家也把这个过程改进成一个端到端训练(end-to-end training) 的过程。你**直接输入这个状态，我们不需要去手工地设计这个特征，就可以让它直接输出动作**。那么就可以用一个神经网络来拟合我们这里的价值函数或策略网络，省去了特征工程 (feature engineering) 的过程。



• 策略函数 (policy function)，agent 会用这个函数来选取下一步的动作。

• 价值函数 (value function)。我们用价值函数来对当前状态进行估价，它就是说你进入现在这个状态，可以对你后面的收益带来多大的影响。当这个价值函数大的时候，说明你进入这个状态越有利。

• 模型 (model)。模型表示了 agent 对这个环境的状态进行了理解，它决定了这个世界是如何进行的。

**深度强化学习**（Deep Reinforcement Learning）：不需要手工设计特征，仅需要输入 State 让系统直接输出 Action 的一个 end-to-end training 的强化学习方法。通常使用神经网络来拟合 value function或者 policy network。

1. policy-based（基于策略的）：Agent 会制定一套动作策略（确定在给定状态下需要采取何种动作），并根据这个策略进行操作。强化学习算法直接对策略进行优化，使制定的策略能够获得最大的奖励。
2. valued-based（基于价值的）：Agent 不需要制定显式的策略，它维护一个价值表格或价值函数，并通过这个价值表格或价值函数来选取价值最大的动作。
3. model-based（有模型结构）：Agent 通过学习状态的转移来采取措施。
4. model-free（无模型结构）：Agent 没有去直接估计状态的转移，也没有得到 Environment 的具体转移变量。它通过学习 value function 和 policy function 进行决策。

**基于策略迭代和基于价值迭代的强化学习方法有什么区别?**

答：

1. 基于策略迭代的强化学习方法，agent 会制定一套动作策略（确定在给定状态下需要采取何种动作），并根据这个策略进行操作。强化学习算法直接对策略进行优化，使制定的策略能够获得最大的奖励；基于价值迭代的强化学习方法，agent 不需要制定显式的策略，它维护一个价值表格或价值

函数，并通过这个价值表格或价值函数来选取价值最大的动作。

2. 基于价值迭代的方法只能应用在不连续的、离散的环境下（如围棋或某些游戏领域），对于行为集合规模庞大、动作连续的场景（如机器人控制领域），其很难学习到较好的结果（此时基于策略迭代的方法能够根据设定的策略来选择连续的动作)；

**有模型（model-based）学习和免模型（model-free）学习有什么区别？**

答：针对是否需要对真实环境建模，强化学习可以分为有模型学习和免模型学习。

1. 有模型学习是指根据环境中的经验，构建一个虚拟世界，同时在真实环境和虚拟世界中学习；
2. 免模型学习是指不对环境进行建模，直接与真实环境进行交互来学习到最优策略。
3. 总的来说，有模型学习相比于免模型学习仅仅多出一个步骤，即对真实环境进行建模。免模型学习通常属于数据驱动型方法，需要大量的采样来估计状态、动作及奖励函数，从而优化动作策略。免模型学习的泛化性要优于有模型学习，原因是有模型学习算需要对真实环境进行建模，并且虚拟世界与真实环境之间可能还有差异，这限制了有模型学习算法的泛化性。

**强化学习中的损失函数**

DL 中的 loss function 目的是使预测值和真实值之间的差距最小, 而 RL 中的 loss function 是是奖励和的期望最大（这点值得商榷，DQN中是目标Q值和现有Q值的方差）。

**蒙特卡罗（回合更新）与时序差分（单步更新）计算价值函数：**

1. 蒙特卡罗用从当前状态到结束状态更新，比如一个任务需要10步完成，第一步有A,B两种策略，如何计算第一步A策略的价值，蒙特卡罗的思想就是（此处暂时不谈折扣系数），采取A策略n（比如说100000次）次，取这n次奖励的均值作为第一步采取A策略的价值。
2. 时序差分是不一定到结束才更新，而是k步更新，比如说是从当前状态到下k步状态更新。

Control 就是说我们去寻找一个最佳的策略，然后同时输出它的最佳价值函数以及最佳策略。

在 MDP 里面，**prediction 和 control** 都可以通过动态规划去解决。要强调的是，这两者的区别就在于，预测问题是给定一个 policy，我们要确定它的 value function 是多少。而控制问题是在没有 policy 的前提下，我们要确定最优的 value function 以及对应的决策方案。实际上，这两者是递进的关系，在强化

学习中，我们通过解决预测问题，进而解决控制问题。

1. 同步备份 (synchronous backup) 是指每一次的迭代都会完全更新所有的状态，这样对于程序资源需求特别大。
2. 异步备份 (asynchronous backup) 的思想就是通过某种方式，使得每一次迭代不需要更新所有的状态，因为事实上，很多的状态也不需要被更新。

Policy evaluation 是说给定一个 MDP 和一个 policy，我们可以估算出它的价值函数。

**怎样进行策略搜索？**

1. 最简单的策略搜索办法就是穷举。假设状态和动作都是有限的，那么每个状态我们可以采取这个A 种动作的策略，那么总共就是 |A||S| 个可能的 policy。那我们可以把策略都穷举一遍，然后算出每种策略的 value function，对比一下就可以得到最佳策略。
2. 但是穷举非常没有效率，所以我们要采取其他方法。搜索最佳策略有两种常用的方法：policy iteration和 value iteration。

**Policy iteration 由两个步骤组成**：policy evaluation 和 policy improvement。

1. 第一个步骤是 policy evaluation，当前我们在优化这个 policy π，在优化过程中得到一个最新的 policy。我们先保证这个 policy 不变，然后去估计它出来的这个价值。给定当前的 policy function 来估计这个 v函数。

（Policy evaluation 是说给定一个 MDP 和一个 policy，我们可以估算出它的价值函数。）

1. 第二个步骤是 policy improvement，得到 v 函数过后，我们可以进一步推算出它的 Q 函数。得到 Q函数过后，我们直接在 Q 函数上面取极大化，通过在这个 Q 函数上面做一个贪心的搜索来进一步改进它的策略。
2. Policy Iteration 分两步，首先进行 policy evaluation，即对当前已经搜索到的策略函数进行一个估值。得到估值过后，进行 policy improvement，即把 Q 函数算出来，我们进一步进行改进。不断重复这两步，直到策略收敛。
3. Value iteration 直接把 Bellman Optimality Equation 拿进来，然后去寻找最佳的 value function，没有 policy function 在这里面。当算出 optimal value function 过后，我们再来提取最佳策略。
4. （一个是给状态迭代最优策略，一个是给状态最优价值）

Q 函数 (action-value function)：其定义的是某一个状态某一个行为，对应的它有可能得到的 return的一个期望（over policy function）。

1. MDP 中的 prediction（即 policy evaluation 问题）：给定一个 MDP 以及一个 policy π ，去计算它的value function，即每个状态它的价值函数是多少。其可以通过动态规划方法（Iterative Algorithm）解决。
2. MDP 中的 control 问题：寻找一个最佳的一个策略，它的 input 就是 MDP，输出是通过去寻找它的最佳策略，然后同时输出它的最佳价值函数 (optimal value function) 以及它的这个最佳策略 (optimalpolicy)。其可以通过动态规划方法（Iterative Algorithm）解决。
3. 最佳价值函数 (Optimal Value Function)：我们去搜索一种 policy π ，然后我们会得到每个状态它的状态值最大的一个情况，v∗ 就是到达每一个状态，它的值的极大化情况。在这种极大化情况上面，我们得到的策略就可以说它是最佳策略 (optimal policy)。optimal policy 使得每个状态，它的状态函数都取得最大值。所以当我们说某一个 MDP 的环境被解了过后，就是说我们可以得到一个 optimal value function，然后我们就说它被解了。

**比较下 TD（差分） 和 MC（蒙特卡罗）**

1. TD 可以在线学习 (online learning)，每走一步就可以更新，效率高。MC 必须等游戏结束才可以学习。
2. TD 可以从不完整序列上进行学习。MC 只能从完整的序列上进行学习。
3. TD 可以在连续的环境下（没有终止）进行学习。MC 只能在有终止的情况下学习。
4. TD 利用了马尔可夫性质，在马尔可夫环境下有更高的学习效率。MC 没有假设环境具有马尔可夫性

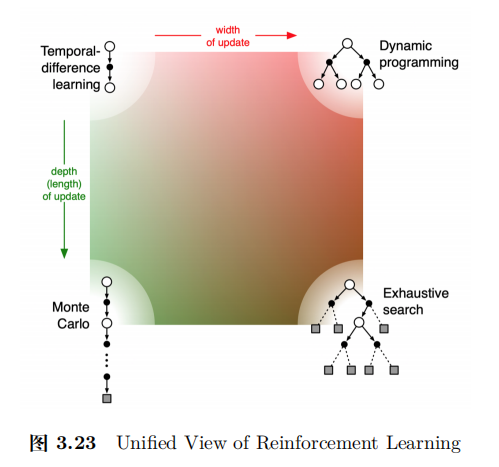
**Bootstrapping**：更新时使用了估计：

1. MC 没用 bootstrapping，因为它是根据实际的 return 来更新。
2. DP 用了 bootstrapping。 TD 用了 bootstrapping。

**Sampling**：更新时通过采样得到一个期望：

1. MC 是纯 sampling 的方法。
2. DP 没有用 sampling，它是直接用 Bellman expectation equation 来更新状态价值的。
3. TD 用了 sampling。TD target 由两部分组成，一部分是 sampling，一部分是 bootstrapping。

四种Q值更新方式



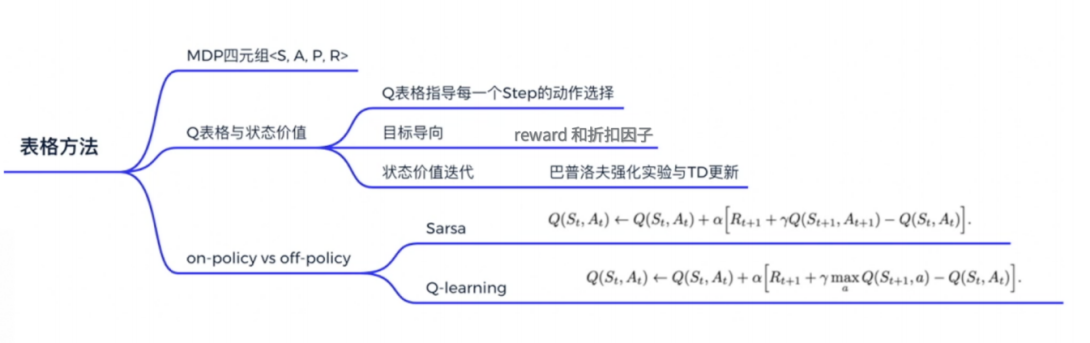
**on-policy和off-policy的区别？**

1. On-policy在学习的过程中，只存在一种策略，它用一种策略去做 action 的选取，也用一种策略去做优化。**一边探索一边学**。
2. Off-policy有两种策略，第一个策略是我们需要去学习的策略，即 target policy(目标策略)，一般用 π 来表示，Target policy就像是在后方指挥战术的一个军师，它可以根据自己的经验来学习最优的策略，不需要去和环境交互。另外一个策略是探索环境的策略，即 behavior policy(行为策略)，一般用 µ 来表示。µ 可以大胆地去探索到所有可能的轨迹，采集轨迹，采集数据，然后把采集到的数据喂给 target policy 去学习。

**Off-policy learning 优点：**

1. 我们可以利用 exploratory policy 来学到一个最佳的策略，学习效率高；
2. 可以让我们学习其他 agent 的行为，模仿学习，学习人或者其他 agent 产生的轨迹；
3. 重用老的策略产生的轨迹。探索过程需要很多计算资源，这样的话，可以节省资源。
4. **learning 是 off-policy 的时序差分学习方法**

**Sarsa是 on-policy 的时序差分学习方法。**



**Tip 1: Add a Baseline**

因为我们做的是采样，有一些动作可能从来都没有采样到。在某一个状态，虽然可以执行的动作有 a/b/c，但你可能只采样到动作 b，你可能只采样到动作 c，你没有采样到动作 a。但现在所有动作的奖励都是正的，所以根据这个式子，它的每一项的概率都应该要上升。你会遇到的问题是，因为 a 没有被采样到，其它动作的概率如果都要上升，a 的概率就下降。所以 a 不一定是一个不好的动作，它只是没被采样到。但只是因为它没被采样到，它的概率就会下降，这个显然是有问题的，要怎么解决这个问题呢？你会希望你的奖励不要总是正的。

为了解决奖励总是正的这个问题，你可以把奖励减掉一项叫做 b，这项 b 叫做 baseline。你减掉这项b 以后，就可以让 R(τ n) − b 这一项有正有负。所以如果得到的总奖励 R(τ n) 大于 b 的话，就让它的概率上升。如果这个总奖励小于 b，就算它是正的，正的很小也是不好的，你就要让这一项的概率下降。如果R(τ n) < b ，你就要让这个状态采取这个动作的分数下降。这个 b 怎么设呢？一个最简单的做法就是：你把 τ n 的值取期望，算一下 τ n 的平均值

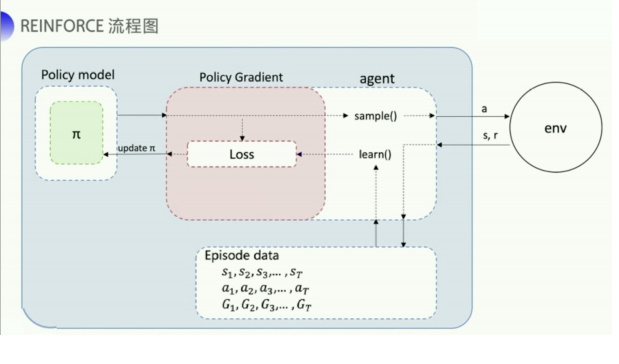
**Tip 2: Assign Suitable Credit**

举个例子，假设这个游戏都很短，只有 3 4 个互动，在 sa 执行 a1 得到 5 分。在 sb 执行 a2 得到 0分。在 sc 执行 a3 得到 -2 分。整场游戏下来，你得到 +3 分，那你得到 +3 分代表在 sb 执行动作 a2 是好的吗？并不见得代表 sb 执行 a2 是好的。因为这个正的分数，主要来自于在 sa 执行了 a1，跟在 sb 执 行 a2 是没有关系的，也许在 sb 执行 a2 反而是不好的，因为它导致你接下来会进入 sc，执行 a3 被扣分，所以整场游戏得到的结果是好的，并不代表每一个行为都是对的。

一个做法是计算这个(s,a)对的奖励的时候，不把整场游戏得到的奖励全部加起来，只计算从这一个动作执行以后所得到的奖励。因为这场游戏在执行这个动作之前发生的事情是跟执行这个动作是没有关系的，所以在执行这个动作之前得到多少奖励都不能算是这个动作的功劳。跟这个动作有关的东西，只有在执行这个动作以后发生的所有的奖励把它加起来，才是这个动作真正的贡献。

把 R − b 这一项合起来，我们统称为优势函数 (advantage function)，用 A 来代表优势函数。优势函数取决于 s 和 a，我们就是要计算的是在某一个状态 s 采取某一个动作 a 的时候，优势函数有多大。优势函数的意义就是，假设我们在某一个状态 st 执行某一个动作 at，相较于其他可能的动作，它有多好。它在意的不是一个绝对的好，而是相对的好，即相对优势 (relative advantage)。因为会减掉一个 b，减掉一个 baseline，所以这个东西是相对的好，不是绝对的好。

独热编码 (one-hot Encoding) 通常用于处理类别间不具有大小关系的特征。例如血型，一共有 4 个取值（A 型、B 型、AB 型、O 型），独热编码会把血型变成一个 4 维稀疏向量，A 型血表示为（1,0,0,0），B 型血表示为（0,1,0,0），AB 型会表示为（0,0,1,0），O 型血表示为（0,0,0,1）。



REINFORCE （基于策略梯度的强化学习的经典算法，其采用回合更新的模式）的流程图。首先我们需要一个 policy model 来输出动作概率，输出动作概率后，我们 sample() 函数去得到一个具体的动作，然后跟环境交互过后，我们可以得到一整个回合的数据。拿到回合数据之后，我再去执行一下 learn() 函数，在 learn() 函数里面，我就可以拿这些数据去构造损失函数，扔给这个优化器去优化，去更新我的 policy model。

近端策略优化 (Proximal Policy Optimization，简称 PPO) 是 policy gradient 的一个变形，它是现在OpenAI 默认的强化学习算法。

可以通过重要性采样把 on-policy 换成 off-policy，但重要性采样（使用另外一种数据分布，来逼近所求分布的一种方法，算是一种期望修正的方法）有一个问题：如果 pθ (at|st) 跟 pθ′ (at|st) 差太多的话，这两个分布差太多的话，重要性采样的结果就会不好。怎么避免它差太多呢？这个就是 Proximal Policy Optimization (PPO) 在做的事情。

PPO 实际上做的事情就是这样，在 off-policy 的方法里要优化的是 Jθ′ (θ)。但是这个目标函数又牵涉到重要性采样。在做重要性采样的时候，pθ (at|st) 不能跟 pθ′ (at|st) 差太多。你做示范的模型不能够跟真正的模型差太多，差太多的话，重要性采样的结果就会不好。我们在训练的时候，多加一个约束 (constrain)。这个约束是 θ 跟 θ′ 输出的动作的 KL 散度 (KL divergence)，简单来说，这一项的意思就是要衡量说 θ 跟 θ′ 有多像。

假设你有两个 actor，它们的参数分别为 θ 和 θ′，所谓参数上的距离就是你算这两组参数有多像。这里讲的不是参数上的距离，而是它们行为上的距离。你先代进去一个状态 s，它会对这个动作的空间输出一个分布。假设你有 3 个动作，3 个可能的动作就输出 3 个值。今天所指的距离是行为距离 (behavior distance)，也就是说，给定同样的状态，输出动作之间的差距。这两个动作的分布都是一个概率分布，所以就可以计算这两个概率分布的 KL 散度。把不同的状态输出的这两个分布的 KL 散度平均起来才是我这边所指的两个 actor 间的 KL 散度。

经典 policy gradient 的大部分时间花在 sample data 处，即当我们的 agent 与环境做了交互后，我们就要进行 policy model 的更新。但是对于一个回合我们仅能更新 policy model 一次，更新完后我们就要花时间去重新 collect data，然后才能再次进行如上的更新。所以我们的可以自然而然地想到，使用 off-policy 方法使用另一个不同的 policy 和 actor，与环境进行互动并用 collect data 进行原先的 policy 的更新。这样等价于使用同一组 data，在同一个回合，我们对于整个的 policy model 更新了多次，这样会更加有效率。

**on-policy 跟 off-policy 的区别是什么？**

答：用一句话概括两者的区别，生成样本的 policy（value-funciton）和网络参数更新时的 policy（valuefunciton）是否相同。具体来说，on-policy：生成样本的 policy（value function）跟网络更新参数时使用的 policy（value function）相同。SARAS 算法就是 on-policy 的，基于当前的 policy 直接执行一次 action，然后用这个样本更新当前的 policy，因此生成样本的 policy 和学习时的 policy 相同，算法为 on-policy 算法。该方法会遭遇探索-利用的矛盾，仅利用目前已知的最优选择，可能学不到最优解，收敛到局部最优，而加入探索又降低了学习效率。epsilon-greedy 算法是这种矛盾下的折衷。优点是直接了当，速度快，劣势是不一定找到最优策略。off-policy：生成样本的 policy（value function）跟网络更新参数时使用的 policy（value function）不同。例如，Q-learning 在计算下一状态的预期收益时使用了 max 操作，直接选择最优动作，而当前 policy 并不一定能选择到最优动作，因此这里生成样本的 policy 和学习时的 policy 不同，即为 off-policy 算法。

**DQN**

**Tip1目标网络**

在状态st，你采取动作 at 以后，你得到奖励 rt ，然后跳到状态 st+1。然后根据这个 Q-function



但是实际上这样的一个输入并不好学习，因为假设这是一个回归问题，Qπ (st, at) 是网络的输出，rt + Qπ (st+1, π (st+1)) 是目标，你会发现目标是会动的。当然你要实现这样的训练，其实也没有问题，就是你在做反向传播的时候，Qπ 的参数会被更新，你会把两个更新的结果加在一起。因为它们是同一个模型 Qπ，所以两个更新的结果会加在一起。但这样会导致训练变得不太稳定，因为假设你把 Qπ (st, at) 当作你模型的输出，rt + Qπ (st+1, π (st+1)) 当作目标的话，你要去拟合的目标是一直在变的，这种一直在变的目标的训练是不太好训练的。

所以你会把其中一个 Q 网络，通常是你会把右边这个 Q 网络固定住。也就是说你在训练的时候，你只更新左边的 Q 网络的参数，而右边的 Q 网络的参数会被固定住。因为右边的 Q 网络负责产生目标，所以叫目标网络。因为目标网络是固定的，所以你现在得到的目标 rt + Qπ (st+1, π (st+1)) 的值也是固定的。因为目标网络是固定的，我们只调左边网络的参数，它就变成是一个回归问题。我们希望模型的输出的值跟目标越接近越好，你会最小化它的均方误差 (mean square error)。

在实现的时候，你会把左边的 Q 网络更新好几次以后，再去用更新过的 Q 网络替换这个目标网络。但它们两个不要一起动，它们两个一起动的话，结果会很容易坏掉。

**Tip2 探索（防止只走最优路径）**

1. **ε-greedy**

我们有 1−ε的概率会按照 Q-function 来决定动作，通常 ε 就设一个很小的值，1 − ε 可能是 90%，也就是 90% 的概率会按照 Q-function 来决定动作，但是你有 10% 的机率是随机的。

1. **Boltzmann Exploration**

这个方法就比较像是策略梯度。在策略梯度里面，网络的输出是一个期望的动作空间上面的一个的概率分布，再根据概率分布去做采样。那其实你也可以根据 Q值去定一个概率分布，假设某一个动作的 Q 值越大，代表它越好，我们采取这个动作的机率就越高。但是某一个动作的 Q 值小，不代表我们不能尝试。

**Tip3 Experience Replay（高效、多样性）**

Experience Replay 会构建一个 Replay Buffer，Replay Buffer 又被称为 Replay Memory。Replay Buffer 是说现在会有某一个策略 π 去跟环境做互动，然后它会去收集数据。我们会把所有的数据放到一个 buffer 里面，buffer 里面就存了很多数据。

这边要注意是 replay buffer 里面的经验可能是来自于不同的策略，你每次拿 π 去跟环境互动的时候，你可能只互动 10000 次，然后接下来你就更新你的 π 了。但是这个 buffer 里面可以放 5 万笔资料，所以 5万笔资料可能是来自于不同的策略。Buffer 只有在它装满的时候，才会把旧的资料丢掉。所以这个 buffer里面它其实装了很多不同的策略的经验。

有了这个 buffer 以后，你是怎么训练这个 Q 的模型呢，怎么估 Q-function？你的做法是这样：你会迭代地去训练这个 Q-function，在每次迭代里面，你从这个 buffer 里面随机挑一个 batch 出来，就跟一般的网络训练一样，你从那个训练集里面，去挑一个 batch 出来。你去采样一个 batch 出来，里面有一把的经验，根据这把经验去更新你的 Q-function。就跟 TD learning 要有一个目标网络是一样的。你去采样一堆 batch，采样一个 batch 的数据，采样一堆经验，然后再去更新你的 Q-function。

**DQN 和 Q-learning 有什么不同？**

整体来说，DQN 与 Q-learning 的目标价值以及价值的更新方式都非常相似，主要的不同点在于：

DQN 将 Q-learning 与深度学习结合，用深度网络来近似动作价值函数，而 Q-learning 则是采用表格存储；

DQN 采用了经验回放的训练方法，从历史数据中随机采样，而 Q-learning 直接采用下一个状态的数据进行学习。

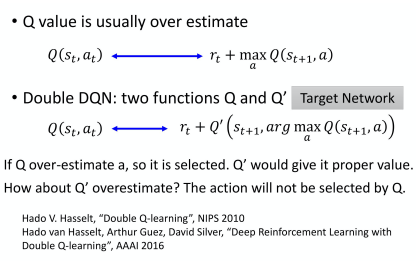
**为什么在 DQN 中采用价值函数近似（Value Function Approximation）的表示方法？**

首先 DQN 为基于深度学习的 Q-learning 算法，而在 Q-learning 中，我们使用表格来存储每一个state 下 action 的 reward，即我们前面所讲的状态-动作值函数 Q(s, a) 。但是在我们的实际任务中，状态量通常数量巨大并且在连续的任务中，会遇到维度灾难的问题，所以使用真正的 Value Function通常是不切实际的，所以使用了价值函数近似（Value Function Approximation）的表示方法。

DOUBLE DQN:

在DQN基础上解决Q值高估的问题。思路是：有两个 Q-network：目标的 Q-network 和你会更新的 Q-network。所以在 DoubleDQN 里面，你会拿你会更新参数的那个 Q-network 去选动作，然后你拿目标网络（固定住不动的网络）去算值。

Double DQN 相较于原来的 DQN 的更改是最少的，它几乎没有增加任何的运算量，连新的网络都不用，因为原来就有两个网络了。你唯一要做的事情只有，本来你在找 Q 值最大的 a 的时候，你是用 Q′来算，你是用目标网络来算，现在改成用另外一个会更新的 Q-network 来算。



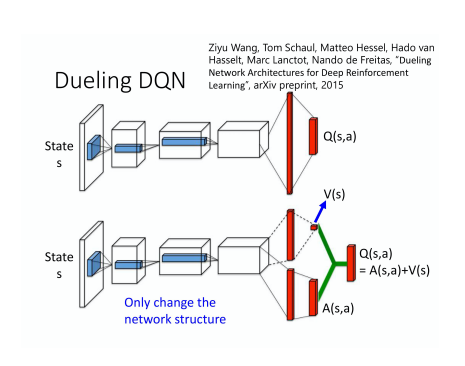
Dueling DQN：

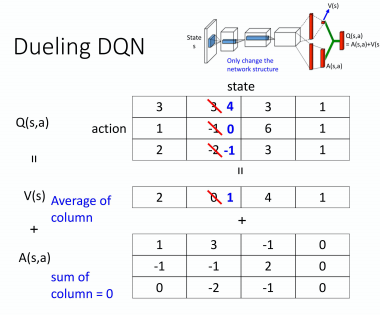
本来的 DQN 就是直接输出 Q 值的值。现在这个 dueling 的 DQN，就是下面这个网络的架构。它

不直接输出 Q 值的值，它分成两条路径去运算：

• 第一条路径会输出一个 scalar，这个 scalar 叫做 V (s)。因为它跟输入 s 是有关系，所以叫做 V (s)， V (s) 是一个 scalar。

• 第二条路径会输出一个 vector，这个 vector 叫做 A(s, a)。下面这个 vector，它是每一个动作都有一个值。你再把这两个东西加起来就可以得到你的 Q 值。



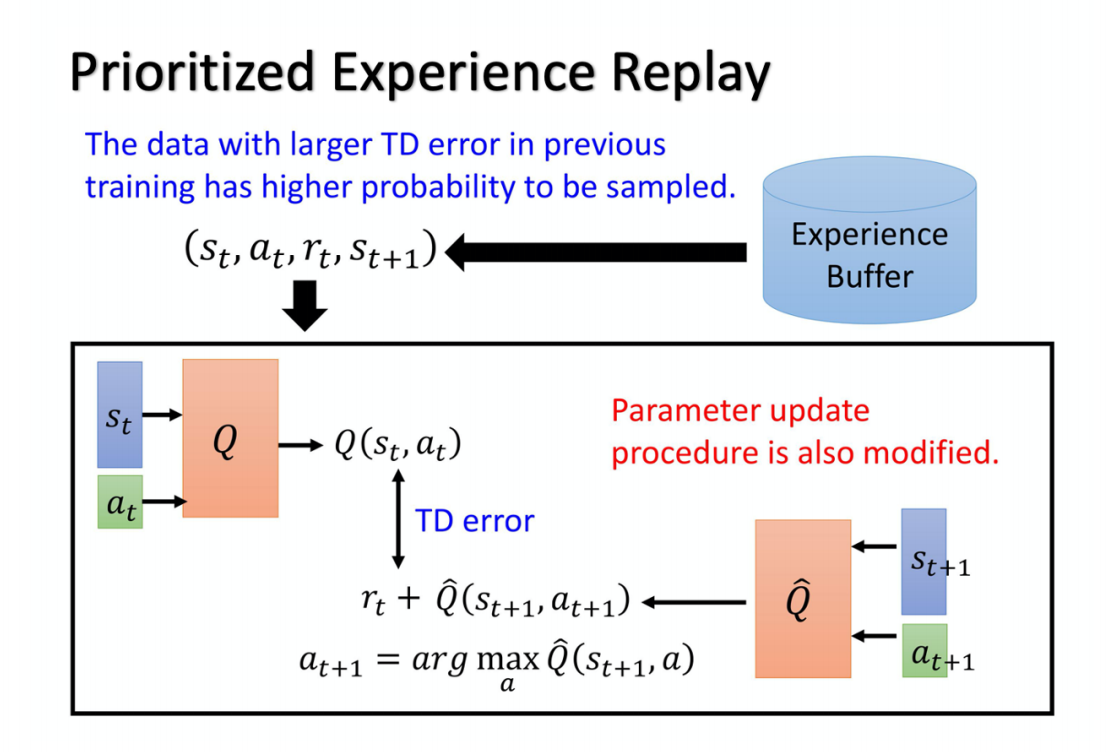


给A添加约束（归一化，每一列平均值为0），这样网络就倾向于去更新V的值。为了避免问题：machine 就学到 V 永远都是 0，然后反正 A 就等于 Q，那你就没有得到任何 Dueling DQN 可以带给你的好处，就变成跟原来的 DQN 一模一样。

**Prioritized Experience Replay：**

我们原来在 sample 数据去训练你的 Q-network 的时候，你是均匀地从 experiencebuffer 里面去 sample 数据。那这样不见得是最好的，因为也许有一些数据比较重要。假设有一些数据，你之前有 sample 过。你发现这些数据的 TD error 特别大（TD error 就是网络的输出跟目标之间的差距），那这些数据代表说你在训练网络的时候，你是比较训练不好的。那既然比较训练不好，那你就应该给它比较大的概率被 sample 到，即给它 priority。这样在训练的时候才会多考虑那些训练不好的训练数据。实际上在做 prioritized experience replay 的时候，你不仅会更改 sampling 的 process，你还会因为更改了

sampling 的过程，更改更新参数的方法。所以 prioritized experience replay 不仅改变了 sample 数据的分布，还改变了训练过程。



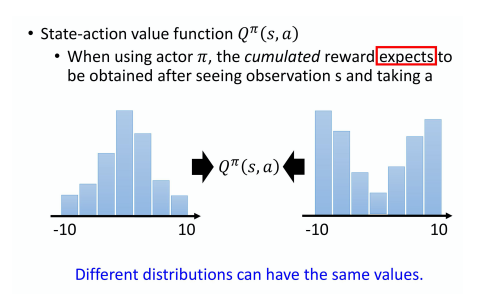
**Balance between MC and TD（全局和单步综合，取n个）**

在 TD 里面，在某一个状态 st 采取某一个动作 at 得到reward rt，接下来跳到那一个状态 st+1。但是我们可以不要只存一个步骤的数据，我们存 N 个步骤的数据。

**Noisy Net（改进探索）**

Noisy Net 的意思是说，每一次在一个 episode 开始的时候，在你要跟环境互动的时候，你就把你的Q-function 拿出来，Q-function 里面其实就是一个网络，就变成你把那个网络拿出来，在网络的每一个参数上面加上一个高斯噪声 (Gaussian noise)，那你就把原来的 Q-function 变成 ˜Q 。因为 ˆQ 已经用过， ˆQ是那个目标网络，我们用 ˜Q 来代表一个 Noisy Q-function。我们把每一个参数都加上一个高斯噪声，就得到一个新的网络叫做 ˜Q。

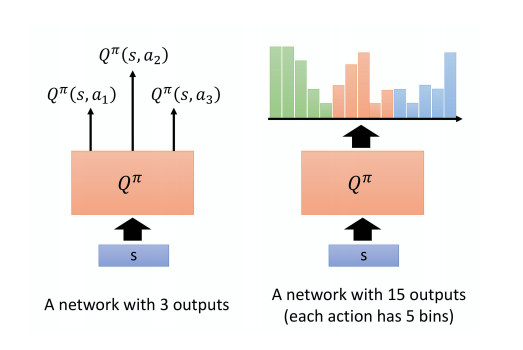
**Distributional Q-function**



累积奖励是一个分布，对它取期望，对它取平均值，你得到了 Q 值。但不同的分布，它们其实可以有同样的平均值。也许真正的分布是右边的分布，它算

出来的平均值跟左边的分布算出来的平均值其实是一样的，但它们背后所代表的分布其实是不一样的。假设我们只用 Q 值的期望来代表整个 reward 的话，其实可能会丢失一些信息，你没有办法 model reward的分布。

Distributional Q-function 它想要做的事情是对分布 (distribution) 建模，怎么做呢？在原来的 Q-function 里面，假设你只能够采取 a1, a2, a3 3 个动作，那你就是输入一个状态，输出 3 个值。3 个值分别代表 3 个动作的 Q 值，但是这个 Q 值是一个分布的期望值。



实际上的做法是说，假设分布的值就分布在某一个 range 里面，比如说 -10 到 10，那把 -10 到 10 中间拆成一个一个的 bin，拆成一个一个的长条图。举例来说，在这个例子里面，每一个动作的 reward 的空间就拆成 5 个 bin。假设 reward 可以拆成 5 个 bin 的话，今天你的 Q-function 的输出是要预测说，你在某一个状态，采取某一个动作，你得到的 reward，落在某一个 bin 里面的概率。

所以其实这边的概率的和，这些绿色的 bar 的和应该是 1，它的高度代表说，在某一个状态采取某一个动作的时候，它落在某一个 bin 的机率。这边绿色的代表动作 1，红色的代表动作 2，蓝色的代表动作3。所以今天你就可以真的用 Q-function 去估计 a1 的分布，a2 的分布，a3 的分布。那实际上在做测试的时候，我们还是要选某一个动作去执行，那选哪一个动作呢？实际上在做的时候，还是选这个平均值最大的那个动作去执行。