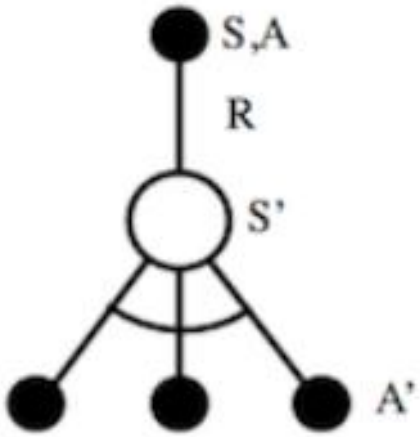
DQN与Policy Network

2019.12

林立晖

Q-learning回顾



Q-target

$$Q(S,A) \leftarrow Q(S,A) + \alpha\left(R + \gamma \max_{a'} Q(S',A') - Q(S,A)\right)$$

收敛时Q ≈ Q-target，Q值更新量非常小

	a1	a2	a3	a4
s1	Q(1,1)	Q(1,2)	Q(1,3)	Q(1,4)
s2	Q(2,1)	Q(2,2)	Q(2,3)	Q(2,4)
s3	Q(3,1)	Q(3,2)	Q(3,3)	Q(3,4)
s4	Q(4,1)	Q(4,2)	Q(4,3)	Q(4,4)

思考：Q矩阵储存了所有的 (s, a) 对所对应的Q值，大小是 num_state x num_state 的，会有什么问题？在黑白棋中需要维护多少个状态？

	a1	a2	a3	a4
s1	Q(1,1)	Q(1,2)	Q(1,3)	Q(1,4)
s2	Q(2,1)	Q(2,2)	Q(2,3)	Q(2,4)
s3	Q(3,1)	Q(3,2)	Q(3,3)	Q(3,4)
s4	Q(4,1)	Q(4,2)	Q(4,3)	Q(4,4)

传统方法中，对每个输入的state，我们都需要有一个对应action，才能计算 (s, a) 对应的Reward，从而计算Q值。当状态非常多时，储存Q矩阵往往是不可行的。即使对Q矩阵使用三元组表示，删除非连通位置，依旧难以应对所有情况。

黑白棋的棋盘大小是19x19，共有361个落子位。每个位置有三个状态，黑，白，或者空。因此一共有3^361种状态（每个状态对应一种具体的棋盘局面）。

价值函数近似
Value Function Approximation

Q矩阵之所以会出现维数灾难，本质上是因为 (s, a) 和Q值是一一映射的，传统方法保存了每一个映射。

解决思路：如果可以找到这些单射共同的函数表达式，就可以不必储存矩阵。
即：找到一个函数 f (s, a; θ) = Q(s, a)，其中θ是函数的参数。

Q矩阵所保存的内容，实际上可以看作是一个三维离散分布的二维点-值表格（Q-table）。因此，我们的真正目的是找到一个**分布函数** f (s, a; θ) 来近似Q值的真实分布。这就是所谓的价值函数近似。

思考：如何确定这个分布函数？

DQN
Deep Q-learning Network

不难想到，拟合一个任意分布的强有力方法就是深度神经网络。一般的做法是只输入状态s (可以是一个高维向量，例如棋盘状态可以表示成一个361维的三值向量)，输出num_action维的向量，每个位置的元素值等于对应action的Q值。

$$L(w) = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta) - Q(s, a; \theta)\right)^2\right]$$

思考:

(1) DQN的标签如何获取?

(2) 在RL中，状态之间是互相依赖的，而DL中的样本默认是相互独立的，如何解决相关性问题?

(3) RL学习到的Q值分布不是固定的（例如两局对弈之间的Q值分布可能差异很大），而DL学习到的Q值分布参数（权重）是固定的，如何解决这个问题?

(4) 有研究表明，使用非线性网络表示值函数时出现不稳定等问题，如何解决?

DQN
Deep Q-learning Network

(1) DQN的标签如何获取?

使用环境返回的Reward（例如在黑白棋中根据场上敌我棋子数计算局面得分）和同一网络所预测的Q-target值作为标签。

(2) 在RL中，状态之间是互相依赖的，而DL中的样本默认是相互独立的，如何解决相关性问题?

储存所有状态转移样本，每次随机抽取一部分进行训练 (打破相关性对网络的影响)。

(3) RL学习到的Q值分布不是固定的（例如两局对弈之间的Q值分布可能差异很大），而DL学习到的Q值分布参数（权重）是固定的，如何解决这个问题?

储存所有状态转移样本，每次随机抽取一部分进行训练 (随机样本相当于把所有局面打散后训练，避免了顺序训练造成的参数局部适应导致参数偏差过大的问题)。

(4) 有研究表明，使用非线性网络表示值函数时出现不稳定等问题，如何解决?

额外使用一个网络来生成Target Q-value（计算Target-Q时不使用Q网络的参数），隔一定时间更新。

DQN训练过程

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

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

$\langle s_1, a_1, r_1, s_2 \rangle$
$\langle s_2, a_2, r_2, s_3 \rangle$
$\langle s_3, a_3, r_3, s_4 \rangle$
$\langle s_4, a_4, r_4, s_5 \rangle$
$\langle s_5, a_5, r_5, s_6 \rangle$
$\vdots$

$$\theta_{t+1} = \theta_t + \alpha \left[r + \gamma \max_{a'} \hat{Q}(s', a'; \theta^{-}) - Q(s, a; \theta) \right] \nabla Q(s, a; \theta)$$

Policy Network

除了Q-learning这类value-based的方法之外，强化学习中另一种非常重要的方法就是Policy Gradients。

Policy Gradients方法直接预测在某个环境下应该采取的action，而Q-learning方法预测某个环境下所有action的期望值 (即Q值)。一般来说，传统Q-learning方法只适合有少量离散取值的action环境，而Policy Gradients方法适合有连续取值的action环境。

主要的区别在于Policy Network的输出是action的概率，并根据这个概率选择action：

$$p(a) = \pi(a|s;\theta)$$

而DQN中使用的是ε-greedy算法进行选择。

Policy Network

Policy Network一个最直接的问题就是如何设计目标函数。

最直接的方法是使用discount reward (衰减回报):

$$L(\theta) = \mathbb{E}\left(r_1 + \gamma r_2 + \gamma^2 r_3 + \ldots | \pi(s;\theta)\right)$$

这个损失函数的值是由环境给出的reward计算得出的，和Policy Network没有直接联系。

一种合理的方法是设计一个评估函数，对每个action进行评价，并乘以该action的负对数似然作为权重。加权求和后就能够得到如下损失函数:

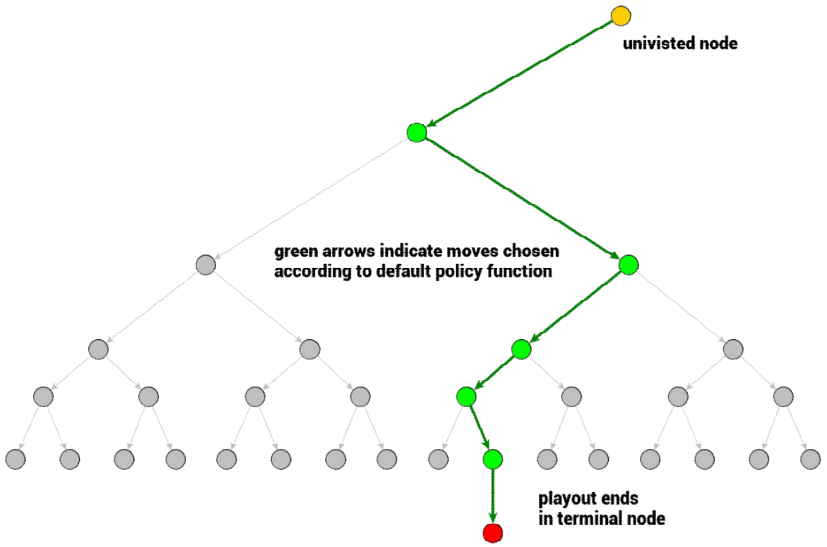
$$L(\theta) = \sum \log \pi(a|s;\theta) f(s,a)$$

评估函数的一个重要作用是判定某个action是“好的”还是“坏的”，如果是前者，则最大化损失函数；后者则最小化损失函数（实际上就是在调整每个action的概率）。更简单地，可以直接用评估函数作为损失函数，结果类似。

Monte Carlo Tree Search (MCTS)

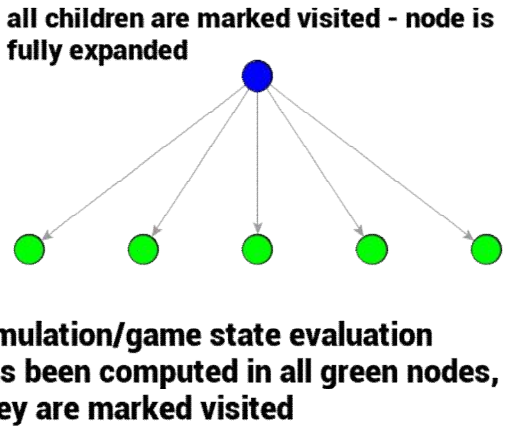
不同于minmax（需要展开整个搜索树）和α-β剪枝（减小搜索空间，效果≤minmax），MCTS在选择下一步策略的时候是基于多次模拟的。

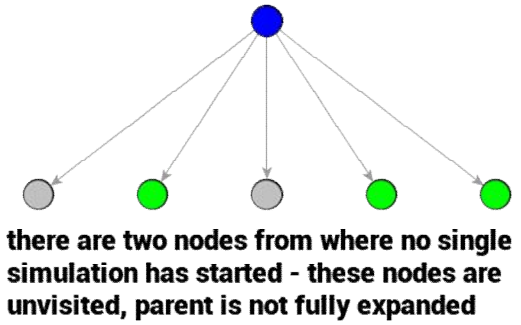
模拟：一次模拟即产生一条从开始节点到终止节点的完整路径。在MTCS中，在节点间的移动方式是基于Rollout Policy的，考虑到效率问题往往使用均匀分布来选择在某节点的下一步移动。模拟的结果可以是任意合法的终止状态（Win，Lose，Tie）。这样能够较快地构建完整的MCT。



Monte Carlo Tree Search (MCTS)

MTCS中的搜索树展开方式，是基于“完全拓展”和“未完全拓展”节点的。如果一个节点的所有子节点都已经被访问过（至少被评估过一次），那么就是完全拓展的节点，反之则是未完全拓展节点。



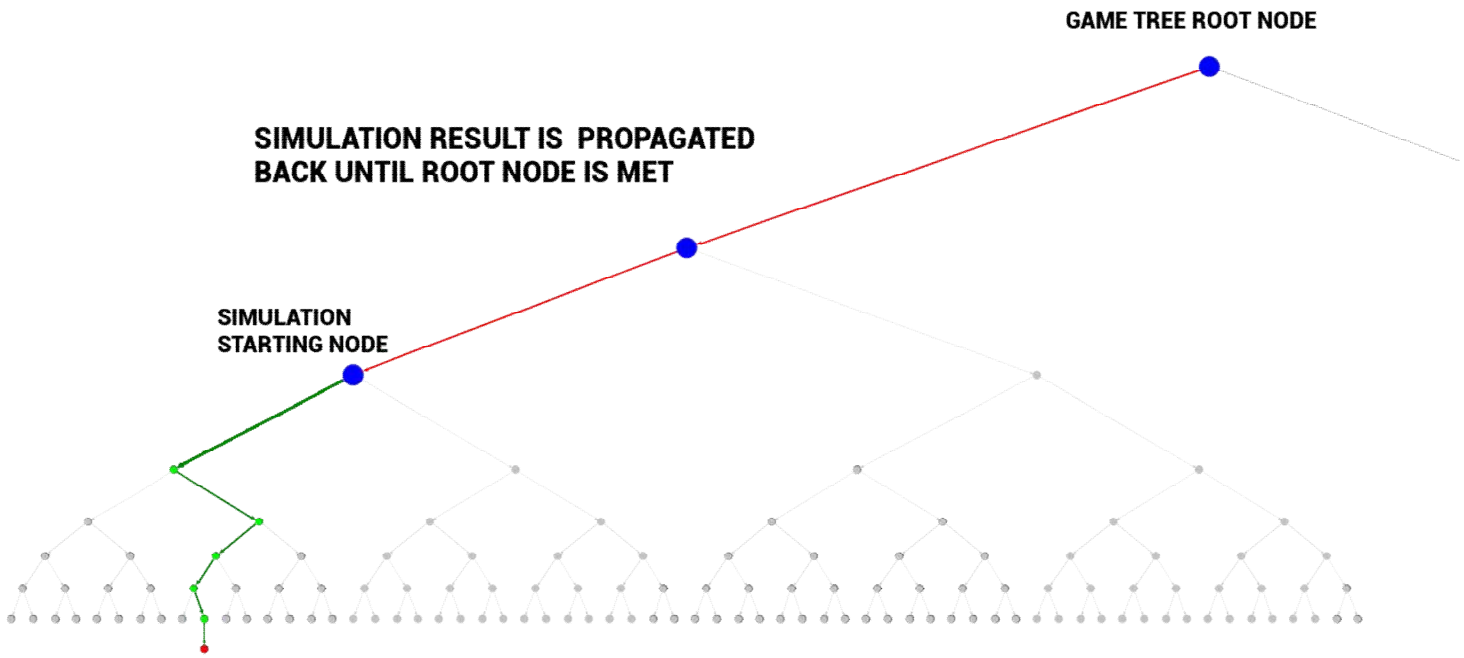


Monte Carlo Tree Search (MCTS)

MTCS通过反向传播更新每个节点的统计信息：

Q(v)：总模拟收益，最简单形式的就是所有考虑的节点的模拟结果之和

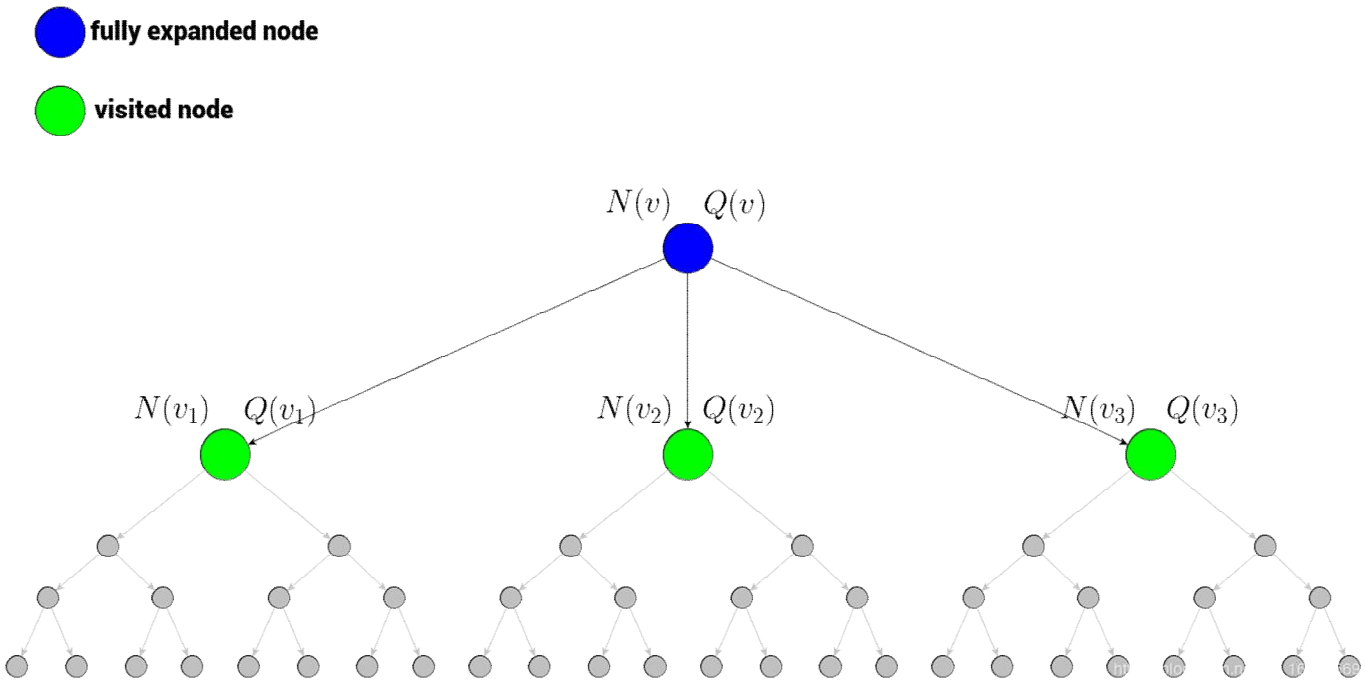
N(v)：总访问次数，代表这个节点有多少次出现在反向传播路径上



Monte Carlo Tree Search (MCTS)

基于树的置信度上界 (Upper Confidence Bound applied to Trees，UCT) 的MCT构建过程:

$$\mathbb{UCT}(v_i, v) = \frac{Q(v_i)}{N(v_i)} + c\sqrt{\frac{\log(N(v))}{N(v_i)}}$$



DQN + Policy Network + MCTS：以AlphaGo为例

Supervised Learning Policy Networks：

CNN + softmax，以人类专家的落子方式作为训练样本，模拟人类走法。通过特征工程，简化网络进行加速，得到Fast Rollout Poilicy Network (FRPN)。

Reinforcement Learning Policy Networks：定义一个reward function (即前文提到的评估函数)，结合加权求和损失函数进行更新：$L(\theta) = \sum \log \pi(a|s;\theta) f(s,a)$。训练得到一个以最大化赢棋概率为目标的策略网络。

Reinforcement Learning Value Networks：首先使用RLPN海量的自我对弈，并根据其策略 (近似最优) 计算出各种局面的估值 (期望收益)，再训练一个value network (DQN等) 来拟合这些估值，从而获得一个能够快速评价各种局面的，泛化能力极强的价值网络。

整合：SLPN用于计算节点下所有可能的action (走法) 的先验概率，根据先验和节点收益估计进行选择; FRPN用于加速选择action (思考时间有限)，RLVN用于对访问到的每一步action 进行估值 (为MCTS提供更优的评估函数)，两者快速模拟博弈，最终选取综合评估最优的action。RLPN主要用于产生海量样本以训练RLVN，在实际中使用模拟人类专家的SLPN能够使得策略更富多样性，更适合AlphaGo中的搜索算法。