Genifer 5.3 theoretical notes

YKY (甄景贤)

August 17, 2015

1 Top-level architecture

在最高层面上，RL 控制 RNN：

可以想像，RNN 是 RL 的 “mental model”；换句话说，我们的 RL 比普通 RL 有更复杂的内部模型。

馀下的工作是，定义 RL 的四个元素：states, actions, rewards, policy。

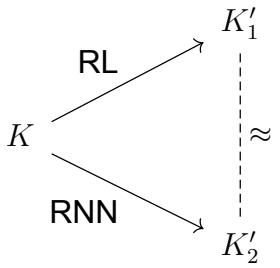
2 统一 RL 和 RNN

从**强化学习**的角度来看，我们需要学习的是这个 policy 函数：

$$\text{policy}: \quad \text{state} \xmapsto{\text{action}} \text{state'}$$

而 K 可以看成是我们的 mental state，那么 RL 的 action 就是将 K 变成 K' 的作用。

在我们的系统中，将 K 变成 K' 有两个方法，分别是 RNN 和 RL：

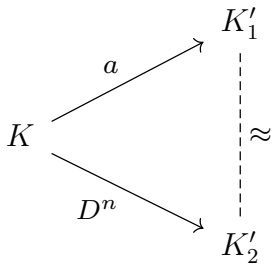


RL 是透过 action a 作用在 K 上，而 RNN 是用 D 作用在 K 上。

注意：RNN 和 RL 都是学习算法，若将他们同时应用在同一问题上，必然会导致 conflicts，除非有方法把两种学习算法合并。

假设学习是正确的话，K'_1 和 K'_2 应该是大致相同的 — 但这忽略了一个可能性，就是其中一条路径经过很多步以后才到达相近的终点。[1]

现在我假设 D 的结构是比较「微细」的，亦即要 D^n 次才等於一个 a：



[1]这情况在 term rewriting systems (TRS) 中已经被研究过；如果在 TRS 中任何两个不同的 rewriting 路径都必然会 converge 到同一结果，则叫做有 Church-Rosser property。例如 Church 发明的 λ-calculus 有这个性质。

用另一个方法表示，a 是 D^n 在 K 点的一个 section（切片）：$a = D^n|_K$。

问题是 RNN 和 RL 这两条路径究竟有没有本质上的分别？

- 它们内部的 representation 不同：RNN 是一个多层的神经网络，但 RL 的 representation 是 $Q(\text{state}, \text{action})$，虽然 Q 本身也可以用神经网络实现。
- RL 是透过 reward 学习，RNN 是透过 error 学习。RL 的适用性较广，因为并不是所有问题都有「正确答案」可以用误差去衡量，但在 RL 中我们只需要在良好行为出现时加以赞赏。
- 或许我们应该用 RL 指导 RNN 的学习（RNN 是更 fine-grain 的）....

从强化学习的角度看，某些推导过程的结果，可以给予奖励：

$$K_0 \overset{a}{\longmapsto} K_\perp \quad \updownarrow \bigstar$$

$\updownarrow \bigstar$ 的意思是「给予正或负奖励」。我们要学习的是 a 也就是 action。学习算法的基础是著名的 Bellman optimality condition (见下节)。

学习的过程可以是 “interlaced”：每做一次 RL，做 n 次 RNN。

RNN 的学习还可以包括 **neurogenesis** (增加 RNN 的神经元)，但我暂时未想到这方面。

有 4 种学习模式：

- 学习 听 / 讲
- RL-based learning
- inductive learning

3 Bellman equation: the basis of RL

回顾一下 Bellman equation 是：

$$U(S) = \max_a \{R(S,a) + \gamma U(S')\}$$

其中 U = utility, R = reward, γ = discount factor, a = action, $S \overset{a}{\mapsto} S'$。它是一条 recursive equation，它表示本状态的最佳效用 $U(A)$，與下一状态的最佳效用 $U(S')$，之間的關係。

在我们的 architecture 里，K 就是 state，作用在 K 上的 action 永远是 D = RNN，但 D 可以變化，$\mathbf{D} \ni D$ 是所有 RNN 的 weights 的空间。

$U : \mathbf{S} \to \mathbb{R}$ 是一个给出 mental states 的效用的函数，这是强化学习中独有的。对了！这就是强化学习和 back-prop 不同的地方！有些 mental states 虽然不是最终答案，但它们的效用相对地比较高。或许我们可以另外建立一个神经网络去 approximate 效用函数 $U(\text{state})$ 或 $Q(\text{state}, \text{action})$。

RL 的做法是：每个 cycle，我们 apply D （例如说，10 次），然后得到新的 K。然后我们计算：

$$\text{action} = \arg\max_a Q(K,a)$$

亦即是根据 Q 值选取最佳 action。注意：函数 Q 是用一个外在的 neural network 近似地学习的，然后我们要找 Q 的 maximum，那本身需要另一个 optimization process（见附录），而且必需很快，这可能是整个系统的瓶颈。

至於 $Q(K,a)$ 的学习，基本的 update rule 是：

$$Q(K,a) \;\texttt{+=}\; \eta(R(K,a) + \gamma \max_{\hat{a}} Q(K',\hat{a}))$$

其中 η 控制学习速度。

離題一點：Bellman 方程有**微分版本**，叫 Hamilton-Jacobi 方程，现在两者可以统一为 Hamilton-Jacobi-Bellman 方程。

定义连续的 “utility”：

$$U(x,t) = \min_u \{ \int_t^{t'} C(x,u)dt + U(x',t') \}$$

其中 t 是时间，u 是 control parameters，C 是 cost-rate function：

$$\int C dt = R = \text{reward}$$

这个积分表示「路径中的 cost」，$U(x',t')$ 是「终点的 cost」。

这条方程写成微分形式就是 Hamilton-Jacobi 方程：

$$\frac{d}{dt}U(x,t) = \min_u \{ C(x,u) + \langle \nabla U(x,t), f(x,u) \rangle \}$$

而 x 必需遵从：

$$\dot{x}(t) = f(x(t), u(t))$$

这个方程的的形式很接近量子力学中的 Schrödinger 方程：

$$i\hbar \frac{\partial}{\partial t}\Psi(x,t) = \left[V(x,t) + \frac{-\hbar^2}{2\mu}\nabla^2 \right] \Psi(x,t)$$

Ψ 类似於我们的 U（可能它是一个自然界想 minimize 的东西?）

4 学习 听 / 讲

我发现 听 / 讲 是可以靠 RL 学习的:

1. 学习听:

$$\overset{\checkmark}{\text{输入句子}} \xrightarrow{?} K \xrightarrow{\text{讲}} \text{输出句子} \approx_L \overset{\checkmark}{\text{testers}}$$

2. 学习讲:

$$\overset{\checkmark}{\text{输入句子}} \xrightarrow{\text{听}} K \xrightarrow{?} \text{输出句子} \approx_L \overset{\checkmark}{\text{testers}}$$

✓ 代表已知的 data （= training set），? 是我们想学习的 map，红色代表 inter-dependence （听和讲的 maps 在学习中互相依赖），$\approx_L$ 比较两句句子的表面近似度，它是外在的 function。Testers 是一些自然语言句子，用来试验听和讲的正确性。

这似乎是一个如何设计 training regimen 的问题多於算法的问题。

算法方面，举例来说，假设 K 已经是 prepared，

$$\overset{\checkmark}{K} \xrightarrow{D^n} \text{输出句子} \quad \updownarrow \bigstar$$

....

5 Inductive learning

以前已经讲过，用 back-prop 做，没有特别困难。Back-prop 和 RL 是独立的。

Appendix: Finding maximum in a back-prop NN

Problem statement: We have used back-prop to train a neural network to approximate $Q(K, a)$. But we don't have the explicit form of Q; Q is realized *implicitly* in the NN. Now we need to find $\arg\max_a Q(K, a)$.

We can use **gradient descent** to find the (local) maxima, and repeat this over randomly-chosen starting points. After many trials the result will approach global maximum.

The gradient descent **update rule** is:

$$x^{k+1} = x^k - \eta \nabla y(x^k).$$

This process is quite similar to back-prop, with the difference that back-prop uses the gradient $\frac{\partial \mathcal{E}}{\partial W}$ whereas here we use the gradient $\frac{\partial y}{\partial x}$.

For each layer,

$$y_j = \sigma(\sum W_{ji} x_i)$$

and the overall output is:

$$y = y_J \circ y_{J-1} \circ .... y_0(x).$$

J is the number of hidden layers.

We then calculate

$$\nabla y = \left[\frac{\partial y}{\partial x_l}\right] = \frac{\partial y_J}{\partial y_{J-1}} \frac{\partial y_{J-1}}{\partial y_{J-2}} .... \frac{\partial y_0}{\partial x_l}$$

At each step, we move a small amount in this direction. But the above direct calculation of ∇y seems very complicated. An alternative is to use **numerical differentiation**.