Genifer 5.3 theoretical notes

YKY (甄景贤)

August 19, 2015

Abstract

This is a completely new architecture: a long vector represents the cognitive state of the Reasoner, where a "recurrent" neural network acts on it to yield the new cognitive state. This corresponds to a logical inference step in the classical paradigm. A top-level reinforcement learner controls this Reasoner, having access to its cognitive states.

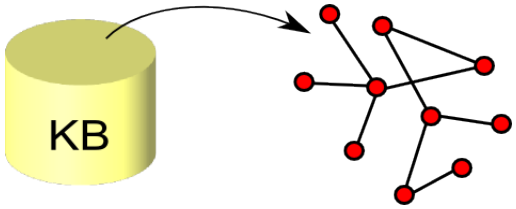
1 Cognitive states

As an example consider:

- It is 3:00AM in the morning
- I am hungry
- The refrigerator is empty
- The MacDonald's downstairs is closed
- .... etc

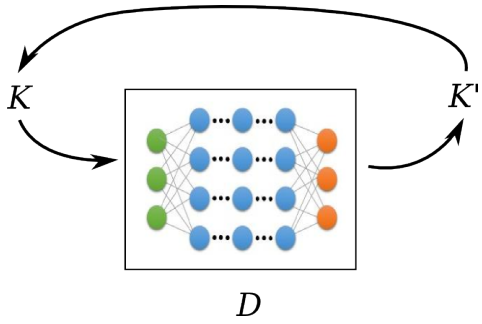
This is an example of my **cognitive state** that can be acted on to generate new conclusions.

In classical logic-based AI we have this dynamics:



where the KB (knowledge base) contains logical **facts** and **rules**. *Rules act on facts* to generate new **propositions** about the world.

In the new Genifer architecture, we introduce the **cognitive state vector** K which is acted on by a **recurrent neural network** (RNN) [1] D:

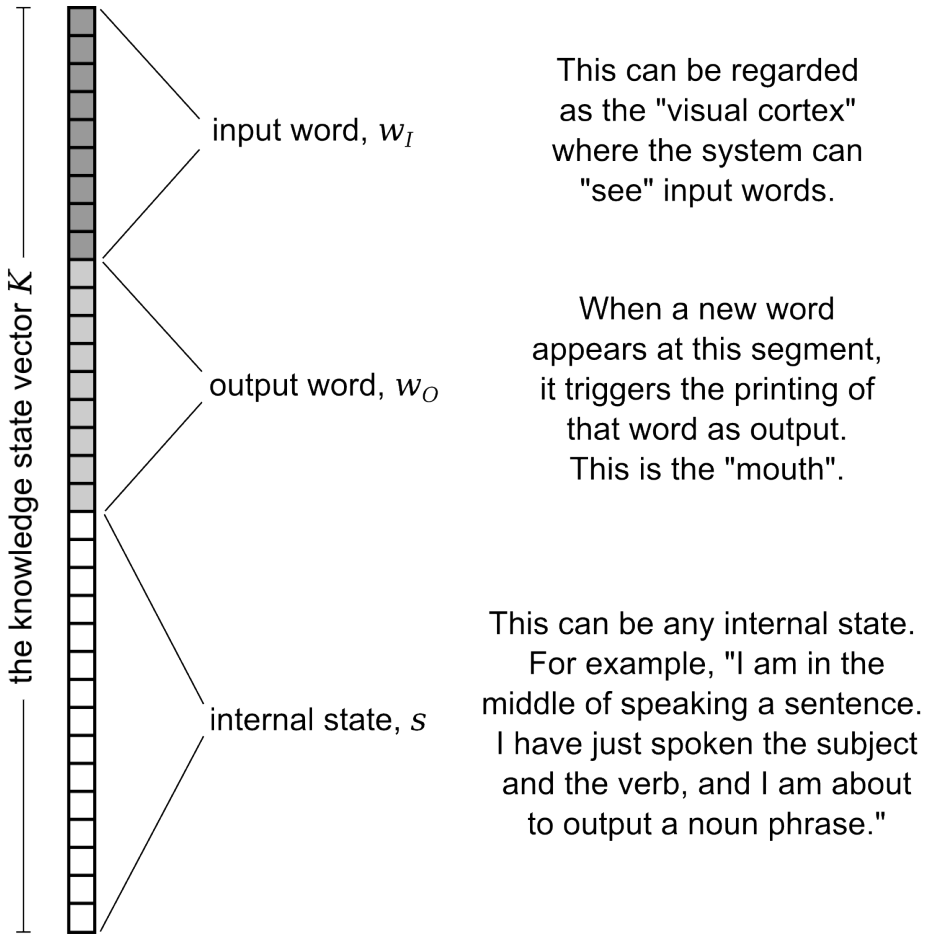


K can be regarded as "**working memory**" in cognitive science. D stands for "deduction" in classical logic-based AI, but as we shall see it is more general than deduction. The key point is the ability of D to perform *multi-step* inference operations on K.

[1] Recurrent here means a simple multi-layer feed-forward NN with its output fed back to its input, which is different from the standard meaning where hidden neurons have *free* connections.

facts in KB	$\Leftrightarrow$	cognitive state vector K
logic rules	$\Leftrightarrow$	recurrent neural network

The cognitive state vector K could be like this:



The general operation of Genifer could be like this:

$$\text{sentence input} \xrightarrow{\mathcal{L}} \mathbf{K} \xrightarrow{\mathcal{L}^{-1}} \text{sentence output}, \quad \mathbf{K} \xrightarrow{D} \mathbf{K}$$

where $\mathcal{L}$ is the "**natural-language map**". As we shall see, $\mathcal{L}$ is not really needed.

The use of an RNN to act on the cognitive state vector is a radical departure from classical logic-based AI. There is nothing in the new system that directly corresponds to propositions in classical logic. For example, Genifer may believe that "I should not be superstitious", but there may be nothing in the RNN or in K that corresponds to the *symbolic* representation of such a sentence. If Genifer utters the sentence "I should not be superstitious", it is *generated* from the cognitive dynamics of the RNN.

Moreover, D no longer needs to correspond to *single* steps of logical inference. Our formulation implicitly relaxed the requirement so that D could be "**fractional**" or even "**infinitesimal**" inference steps. The latter case might involve Lie groups or algebras, something I have not explored yet.

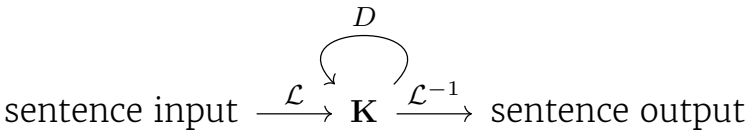
2 Language map

Genifer is not the first model that uses an RNN to model logical reasoning. My friend Joseph Cheng showed me a paper from Facebook AI Research group [1], that proposed **Memory Networks** for Q & A tasks. It has a component I = input feature map, responsible for parsing natural-language sentences into internal feature representations.

$\mathcal{L}$ turns NL sentences into internal representation of knowledge in $\mathbf{K}$, where $\mathbf{K} \ni K$ is the space of all cognitive states:

$$\begin{aligned} \mathcal{L}: &\quad \mathbf{S} &&\to \mathbf{K} \\ \mathcal{L}: &\quad \text{sentence} &&\mapsto K \end{aligned}$$

(Actually the sentence representation is just a part of K.)



Having this **language map** is very convenient, but it has a few problems:

- it requires NLP parsing, which is troublesome; better if we could avoid it
- the map $\mathcal{L}$ basically *fixed* the internal knowledge representation format. But my intuition is that the knowledge representation should better be "unknown", it should be *induced* through the learning process of D.

  In traditional logic-based systems, K = KB is a collection of logical propositions, $K = \bigsqcup P_i$. At that time, we tried to organize the KB hierarchically, so as to speed up searching and retrieval. But I feel that if the structure of **K** is also the same, it would be "too similar" to the original setting, and everything is "too orderly", which may not be the best way to apply neural networks.
- Natural langauge requires slow "absorption" or "comprehension", but this process is ignored in the Memory Network model. Translation of NL sentences into logical form (ie, the internal representation) can almost be done *instantaneously*. But if we input the raw text of a *World History* book to Genifer, and she translated it into internal representation within 1 second, can we say that Genifer has truly "understood" the content of the book?

3 Slow comprehension

Therefore $\mathcal{L}$ is not an ordinary map but a very complex *process*.

The "slow absorption" of new input consists of these operations:

- **consequences** (find out all logical entailment of the new input, at least within n steps of inference)
- **consistency** (new beliefs do not contradict old ones)
- **explanations** (new beliefs can be explained in terms of old ones)

The process of "comprehension" can be realized as the transformation of K into a "knowing" state K' after n steps of inference:

$$K \overset{D^n}{\longmapsto} K'$$

and we want K' to possess the desired properties (consequence, coherence, explanation). The question is how to test if K' has such properties? Particular difficulty stems from the fact that the representation of $K' \in \mathbf{K}$ is *not transparent*.

To test the content of K' we need a query Q:

$$K \overset{D^n}{\longmapsto} K' \overset{?}{=} Q$$

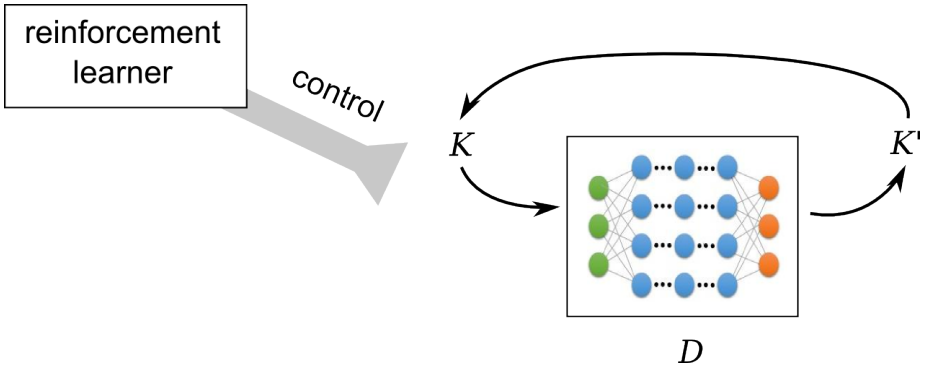
If we have $\mathcal{L}$, we can directly ask in natural language, via the query $Q = \mathcal{L}(\text{sentence})$, to test K. In other words, the advantage of $\mathcal{L}$ is to allow *direct access* to K, so that we can read out or write to the cognitive state. But a simple $\mathcal{L}$ map does not exist in reality.

But all is not lost: One advantage of a neural network is that it can simultaneously learn 2 processes, even if the 2 processes are *inter-dependent*! In §7 we look at how to learn to listen and talk.

4 Top-level architecture

According to Richard Sutton, **reinforcement learning** (RL) should be the controlling module of an AI agent at the top level. So we let the RL control the RNN (recurrent neural network):

Or more detailedly:



We can think of the RNN as the RL's "mental model"; in other words, our RL has a more complex **world model** than traditional RL agents.

See my blog post for an introduction to reinforcement learning.

The remaining work is to define the 4 elements of RL: **states**, **actions**, **rewards**, **policy**.

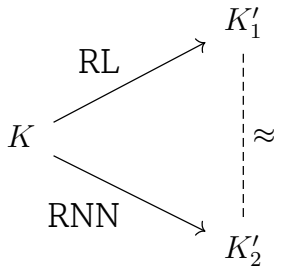
5 Unifying RL and RNN

From the viewpoint of reinforcement learning, we aim to learn the **policy** function:

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

Where K can be regarded as the **mental state**, and thus an **action** in RL is to turn K into K'.

In our system, there are 2 pathways to act on K, via RNN and RL respectively:



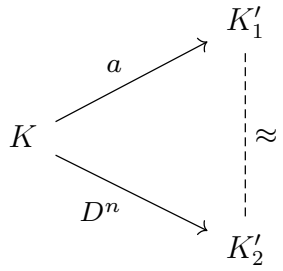
In RL, the action a acts on K, whereas in RNN, D acts on K.

Note: RNN and RL are learning algorithms, and if they are both applied to the same problem, conflicts will necessarily arise, unless there is a way to combine them.

Assuming the learning is correct, K_1' and K_2' should be roughly the same — but this ignored the possibility that one path may take multiple steps to converge with the other path. [2]

[2] This situation has been encountered in term rewriting systems (TRS): If in a TRS any 2 different rewriting paths always converge to the same result, it is said to have the **Church-Rosser property**. For example the λ-calculus invented by Church has this property.

Now I stipulate that D be more "refined", that is to say, applying D^n times may be equivalent to applying a once:



Using a different notation, a is the **restriction** or **section** of D^n at point K: $a = D^n|_K$.

Now the question is, do the RNN and RL paths have any *essential* difference?

- Their internal **representations** are different:
  — RNN is a multi-layer neural network
  — RL's representation is $Q(\text{state}, \text{action})$, although Q could be approximated by a neural network as well.
- RL learns through **rewards**, RNN learns from **errors**. Thus RL has broader applicability, because not all questions have "correct answers" that could be measured by errors. In RL we just need to praise Genifer whenever she displays good behavior.
- The internal cognitive state K exists because of RNN: it is simply the vector input and output of the RNN. Without this K, RL would be clueless as to what are its internal states. It can be said that the RNN provides a *machinery* for RL to control.

From the perspective of reinforcement learning, we could reward some results of multi-step inference:

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

$\updownarrow \bigstar$ means "to give positive or negative rewards". We want to learn a which is the action to be taken at state K. The learning algorithm is based on the famous **Bellman optimality condition** (see next section).

Perhaps we should use RL to *guide* the learning in RNN, as RNN is more fine-grained....

To combine the 2 learning approaches, we could use the technique of **interleaving**: for each step apply RL once, apply RNN n times.

The learning in RNN may also involve **neurogenesis** (adding new neurons and connections), but I have not considered this aspect yet.

There are 4 learning modes:

- learning to listen/talk
- RL-based learning
- inductive learning

6 Bellman equation: the basis of RL

Let's review the Bellman equation:

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

where U = utility, R = reward, γ = discount factor, a = action, $S \overset{a}{\mapsto} S'$. This is a recursive equation, expressing the relation between

A bit of digression: The Bellman equation has a **differential version**, known as the **Hamilton–Jacobi** equation, and nowadays the 2 can be unified as the Hamilton–Jacobi–Bellman equation.

Define a continuous version of "utility":

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

where t is time, u is a set of control parameters, C is the **cost-rate** function:

$$\int Cdt = R = \text{reward}$$

This integral expresses the "cost of the path", whereas $U(x',t')$ is the "cost at termination".

The differential version is the Hamilton–Jacobi equation:

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

where x must obey this dynamics:

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

This equation is very close to the form of the **Schrödinger** equation in quantum mechanics:

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

Ψ is analogous to our U (perhaps it is something that nature wants to minimize?)

the optimal utility of the current state $U(A)$ and the optimal utility of the next state $U(S')$.

In our architecture, K is the state, the action that acts on K is always D = RNN, but D can change, $\mathbf{D} \ni D$ is the space of all the weights in the RNN.

$U : \mathbf{S} \to \mathbb{R}$ is a function returning the **utility** of a mental states, which is *unique* in reinforcement learning. Yes! That is how RL learning is different from back-prop learning! Some mental states could have higher utility than others, even though they are not the final answer state. This corresponds to the feeling where we sometimes know our thinking is on the right track, despite that we have not reached final conclusion.

Perhaps we can use a separate neural network to approximate the utility function $U(\text{state})$ or $Q(\text{state}, \text{action})$.

Algorithm for RL: for each cycle, we apply D (say, 10 times), and we get a new K. Then we calculate:

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

which means we select the best action based on the optimal Q value. Note: the fucntion Q is learned using an external neural network, and then we need to find Q's maximum, which involves *another* optimization process (see appendix). This needs to be very fast, and might be a bottleneck of the entire system.

As for the learning of $Q(K, a)$, the basic **update rule** is:

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

where η controls the learning speed.

7 Learning to listen / talk

Learning to listen/talk can be handled by RL:

1. learning to listen:

$$\text{input } \overset{\checkmark}{\text{sentence}} \xrightarrow{?} K \xrightarrow{\text{talk}} \text{output sentence} \approx_L \text{testers}$$

2. learning to talk:

$$\text{input } \overset{\checkmark}{\text{sentence}} \xrightarrow{\text{listen}} K \xrightarrow{?} \text{output sentence} \approx_L \text{testers}$$

A ✓ means the data is known (= training set), ? is the map we want to learn, red means inter-dependence (the listening and talking functions rely on each other during learning), $\approx_L$ is an *external* function that compares the surface similarity of 2 natural-language sentences. Testers are natural-language sentences used to test the correctness of listening and talking examples.

This seems to be more of a problem of designing training regimens, than of designing algorithms.

8 Inductive learning

For example:
John is Chinese $\wedge$ John wears glasses
Pete is Chinese $\wedge$ Pete wears glasses
Mary is Chinese $\wedge$ Mary wears glasses
Paul is Chinese $\wedge$ Paul wears glasses
Conclusion: all Chinese people wear glasses
Expressed as a logical rule this is:

$$\forall X.\, \text{Chinese}(X) \Rightarrow \text{wear-glasses}(X)$$

but this rule is implicitly encoded in D by the RNN. For example, if K = "Jane is Chinese", then D acting on K would eventually result in K' = "Jane wears glasses".

In other words, given K_0, K^*, learn D such that:

$$K_0 \xrightarrow{D} .... K^*$$

If we have the map $\mathcal{L}$, we could use $K = \mathcal{L}(\text{NL sentence})$ to calculate K_0 and K^*, and this would be feasible.

Learning is done via back-prop, what we need is to start from K_0:

$$\begin{aligned} K_0 &\xrightarrow{D} .... K_\vdash \\ K_\vdash &\geq K^* \end{aligned}$$

but this requires using the $\geq$ relation, discussed below.

The goal is to learn D, to minimize the error $\mathcal{E} = K_\vdash - K^*$.

The calculation of the gradient $\frac{\partial \mathcal{E}}{\partial W}$ should be feasible; here W is the weights of the RNN.

9 Querying

Perhaps querying can be formulated as this problem:

$$\begin{aligned} \text{solve} \qquad D^n(\mathbf{x}) &> K^* \\ K_0 &\geq \mathbf{x} \end{aligned}$$

where $\mathbf{x}$ is the variable. We require $>$ to exceed a threshold ϵ. This is an iterative equation, which seems hopeful.

If D is **monotonous**, ie, $\forall \mathbf{x}\ D(\mathbf{x}) \geq \mathbf{x}$, this fact may be exploited to design faster algorithms.

10 The $\geq$ relation

$\geq$ is the **implication order** in logic (also similar to the **general-specific order**), it has 2 forms:

- all men are mortal $\geq$ Socrates is mortal
- all men are mortal $\geq$ (all men are mortal $\wedge$ the moon is round)

In (topological) vector space theory, we can define a relation $\mathbf{v}_1 \geq \mathbf{v}_2$ between vectors, by selecting an arbitrary **cone** C:

$$\mathbf{v}_1 \geq \mathbf{v}_2 \Leftrightarrow (\mathbf{v}_1 - \mathbf{v}_2) \in C$$

For example, on the 2D plane, C could be the upper-right quadrant.

I am considering: Can we just arbitrarily select a cone in the space $\mathbf{K}$ to define $\geq$, and then let the RNN learn the logical structure of $\geq$ itself (for example, animals $\geq$ cats, $A \geq A \wedge B$, etc)?

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**.

References

[1] Weston, Chopra, and Bordes. Memory networks. *ICLR (also arXiv)*, 2015.