

Wandering in the Labyrinth of Thinking

– a minimalist cognitive architecture combining
reinforcement learning, deep learning, and logic structure

甄景贤 (King-Yin Yan)

General.Intelligence@Gmail.com

Abstract. This paper contains enough details for implementation, and a prototype system is currently under development. We adopt an abstract style of exposition so that the reader can understand there is a large number of variations possible under this architecture.

Keywords: cognitive architecture, reinforcement learning, deep learning, logic-based artificial intelligence

0 Summary

We propose an AGI architecture:

1. With **reinforcement learning** (RL) as top-level framework
 - The external environment is turned “inward”
 - State space = mental space
2. **Logic** structure is imposed on the **knowledge representation** (KR)
 - State transitions are given by logic rules
 - Actions in RL = right-hand side of logic rules
3. The set of logic rules is approximated by a deep-learning neural network (**deep NN**)
 - Logic conjunctions are **commutative**, so the NN should be made **symmetric** using an algebraic trick (§3.1)
 - **Policy-gradient** methods (and variants) may be employed to speed up learning
 - Logic propositions are embedded in “continuous” space, so we have **continuous actions** in RL. The probability distribution over actions can be modeled by **Gaussian kernels** (radial basis functions).

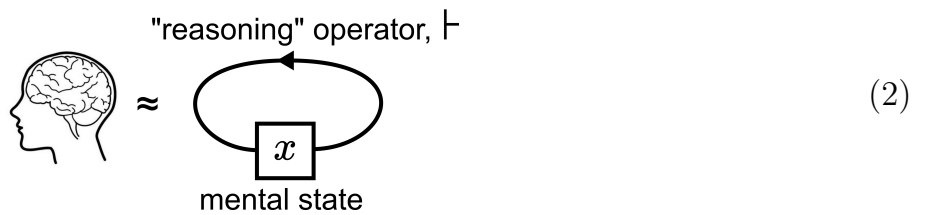
The rest of this paper will explain these design features in detail.

1 Reinforcement-learning architecture

The **metaphor** in the title of this paper is that of RL controlling an autonomous agent to navigate the maze of “thoughts space”, seeking the optimal path:



The main idea is to regard “thinking” as a **dynamical system** operating on **mental states**:



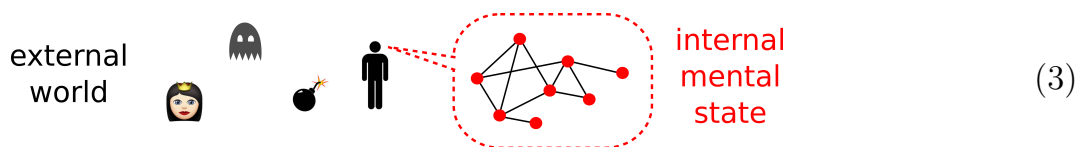
A mental state is a **set of propositions**, for example:

- I am in my room, writing a paper for AGI-2019.
- I am in the midst of writing the sentence, “I am in my room, ...”
- I am about to write a gerund phrase “writing a paper...”

Thinking is the process of **transitioning** from one mental state to another. As I am writing now, I use my mental states to keep track of where I am at within the sentence’s syntax, so that I can construct my sentence grammatically.

1.1 “Introspective” view of reinforcement learning

Traditionally, RL deals with acting in an *external* environment; value / utility is assigned to *external* states. In this view, the *internal* mental state of the agent may change without any noticeable change externally:

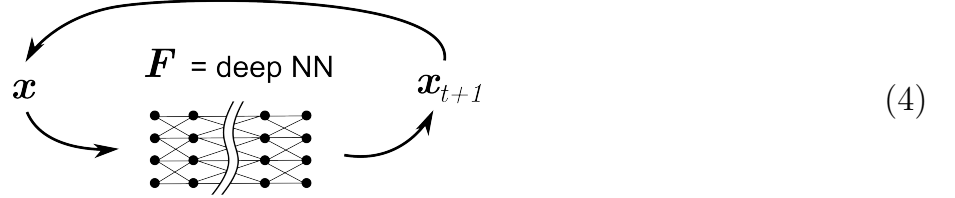


1.2 Actions = cognitive state-transitions = “thinking”

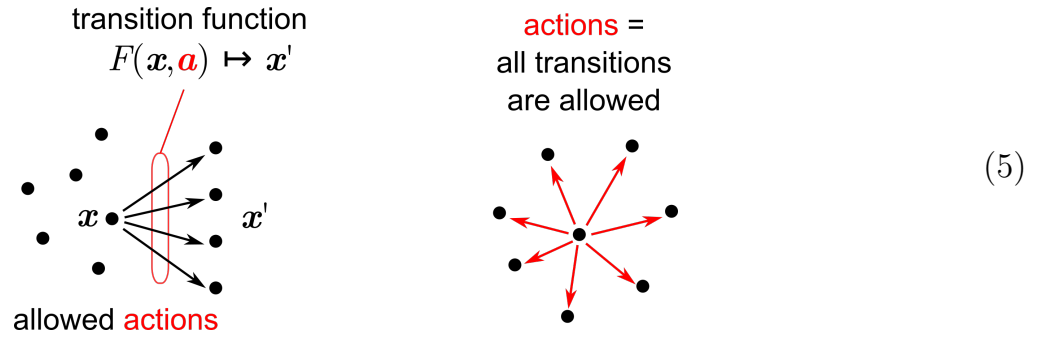
Our system consists of two main algorithms:

1. Learning the transition function \vdash or $\mathbf{F} : \mathbf{x} \mapsto \mathbf{x}'$. \mathbf{F} represents the **knowledge** that constrains thinking. In other words, the learning of \mathbf{F} is the learning of “static” knowledge.
2. Finding the optimal trajectory of the state \mathbf{x} . This corresponds to optimal “thinking” under the constraints of static knowledge.

In our architecture, \mathbf{F} can implemented as a simple feed-forward neural network (where “deep” simply means “many layers”):



In traditional reinforcement learning (left view), the system chooses an action \mathbf{a} , and the transition function \mathbf{F} gives the probability of reaching each state \mathbf{x} given action \mathbf{a} . In our model (right view), all possible cognitive states are potentially **reachable** from any other state, and therefore the action \mathbf{a} coincides with the next state \mathbf{x}' .



1.3 Comparison with AIXI

AIXI’s environmental setting is the same as ours, but its agent’s internal model is a universal Turing machine, and the optimal action is chosen by maximizing potential rewards over all programs of the UTM. In our (minimal) model, the UTM is restricted to a neural network, where the NN’s **state** is analogous to the UTM’s **tape**, and the optimal weights (program) are found via Bellman optimality.

1.4 Infinite-dimensional control

The cognitive state is a vector $\mathbf{x} \in \mathbb{X}$ where \mathbb{X} is the space of all possible cognitive states, the reasoning operator \vdash or \mathbf{F} is an **endomorphism** (an **iterative map**) $\mathbb{X} \rightarrow \mathbb{X}$.

Mathematically this is a **dynamical system** that can be defined by:

$$\begin{array}{ll} \boxed{\text{discrete time}} & \mathbf{x}_{t+1} = \mathbf{F}(\mathbf{x}_t) \\ \text{or } \boxed{\text{continuous time}} & \dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}) \end{array} \quad \begin{array}{l} (6) \\ (7) \end{array}$$

where \mathbf{f} and \mathbf{F} are different but related ¹. For ease of discussion, sometimes I mix discrete-time and continuous-time notations.

A **control system** is a dynamical system added with the control vector $\mathbf{u}(t)$:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), t) \quad (8)$$

The goal of control theory is to find the optimal $\mathbf{u}^*(t)$ function, such that the system moves from the initial state \mathbf{x}_0 to the terminal state \mathbf{x}_\perp .

A typical control-theory problem is described by:

$$\boxed{\text{state equation}} \quad \dot{\mathbf{x}}(t) = \mathbf{f}[\mathbf{x}(t), \mathbf{u}(t), t] \quad (9)$$

$$\boxed{\text{boundary condition}} \quad \mathbf{x}(t_0) = \mathbf{x}_0, \mathbf{x}(t_\perp) = \mathbf{x}_\perp \quad (10)$$

$$\boxed{\text{objective function}} \quad J = \int_{t_0}^{t_\perp} L[\mathbf{x}(t), \mathbf{u}(t), t] dt \quad (11)$$


and we seek the optimal control $\mathbf{u}^*(t)$.

According to control theory, the condition for **optimal path** is given by the Hamilton-Jacobi-Bellman equation:

$$\boxed{\text{Hamilton-Jacobi-Bellman}} \quad 0 = \frac{\partial J^*}{\partial t} + \min_u H \quad (12)$$

$$\frac{d}{dt} V(x, t) = \min_u \{C(x, u) + \langle \nabla V(x, t), f(x, u) \rangle\} \quad (13)$$

1.5 Reinforcement learning / dynamic programming

Reinforcement learning is a branch of machine learning that is particularly suitable for controlling an **autonomous agent** who interacts with an **environment**. It uses **sensory perception** and **rewards** to continually modify its **behavior**. The exemplary image you should invoke in mind is that of a small insect that navigates a maze looking for food and avoiding predators: 

A reinforcement learning system consists of a 4-tuple:

$$\boxed{\text{reinforcement learning system}} = (\mathbf{x} \in \text{States}, \mathbf{u} \in \text{Actions}, R = \text{Rewards}, \pi = \text{Policy}) \quad (14)$$

For details readers may see my *Reinforcement learning tutorial* [?].

U is the total rewards of a sequence of actions:

$$\begin{array}{ccc} \text{total value of state 0} & & \text{reward at time } t \\ & \searrow & \nearrow \\ U(\mathbf{x}_0) & = & \sum_t R(\mathbf{x}_t, \mathbf{u}_t) \end{array} \quad (15)$$

¹ They are related by: $\mathbf{x}(t+1) = \mathbf{F}(\mathbf{x}(t))$, $\mathbf{x}^{-1}(\mathbf{x}(t)) = t = \int_{\mathbf{x}_0}^{\mathbf{x}_t} \frac{d\mathbf{x}}{\mathbf{f}(\mathbf{x}(t))}$, and $f(\mathbf{x}) = \frac{1}{(\mathbf{x}^{-1})'(\mathbf{x}(t))}$. So we can just solve the functional equation $\mathbf{x}^{-1}(\mathbf{F}(\mathbf{x})) - \mathbf{x}^{-1}(\mathbf{x}) = 1$. See [?] §8.2.3.

For example, the value of playing a chess move is not just the immediate reward of that move, but includes the consequences of playing that move (eg, greedily taking a pawn now may lead to checkmate 10 moves later). Or, faced with delicious food, some people may choose not to eat, for fear of getting fat.

The goal of **reinforcement learning** is to learn the **policy function**:

$$\text{policy} : \text{state} \xrightarrow{\text{action}} \text{state}' \quad (16)$$

when we are given the **state space**, **action space**, and **reward function**:

$$\text{reward} : \boxed{\text{state}} \times \boxed{\text{action}} \rightarrow \mathbb{R} \quad (17)$$

The action a is the same notion as the control variable u in control theory.

The central idea of **Dynamic programming** is the **Bellman optimality condition**, which says: “if we cut off a tiny bit from the endpoint of the optimal path, the remaining path is still an optimal path between the new endpoints.”

$$\begin{array}{ccccc} \text{value of entire path} & & \text{reward of choosing } \mathbf{u} \text{ at current state} & & \text{value of rest of path} \\ & \searrow & & \swarrow & \\ \boxed{\text{Bellman equation}} & & U^*(\mathbf{x}) = \max_{\mathbf{u}} \{ R(\mathbf{u}) + U^*(\mathbf{x}_{t+1}) \} & & \end{array} \quad (18)$$

This seemingly simple formula is the entire content of dynamic programming; What it means is that: When seeking the path with the best value, we cut off a bit from the path, thus reducing the problem to a smaller problem; In other words, it is a **recursive relation** over time.

In AI reinforcement learning there is an oft-employed trick known as Q -learning. Q value is just a variation of U value; there is a U value for each state, and Q is the **decomposition** of U by all the actions available to that state. In other words, Q is the utility of doing action \mathbf{u} in state \mathbf{x} . The relation between Q and U is:

$$U(\mathbf{x}) = \max_{\mathbf{u}} Q(\mathbf{x}, \mathbf{u}) \quad (19)$$

The advantage of Q is the ease of learning. We just need to learn the value of actions under each state. This is so-called “**model free learning**”.

The **Bellman equation** governs reinforcement learning just as in control theory:

$$\boxed{\text{optimal path}} = \text{choose max reward on current path segment} + \boxed{\text{the rest of optimal path}} \quad (20)$$

In math notation:

$$U_t^* = \max_{\mathbf{u}} \{ \boxed{\text{reward}(\mathbf{u}, \mathbf{t})} + U_{t-1}^* \} \quad (21)$$

where U is the “long-term value” or **utility** of a path.

1.6 Connections with Hamiltonian and quantum mechanics

This section is optional.

In **reinforcement learning**, we are concerned with two quantities:

- $R(\mathbf{x}, \mathbf{u}) = \text{reward}$ of doing action \mathbf{u} in state \mathbf{x}
- $U(\mathbf{x}) = \text{utility}$ or **value** of state \mathbf{x}

Simply put, **utility** is the integral of instantaneous **rewards** over time:

$$\boxed{\text{utility } U} = \int \boxed{\text{reward } R} dt \quad (22)$$

In **control-theoretic** parlance, it is usually defined the **cost functional**:

$$\boxed{\text{cost } J} = \int L dt + \Phi(\mathbf{x}_{\perp}) \quad (23)$$

where L is the **running cost**, ie, the cost of making each step; Φ is the **terminal cost**, ie, the value when the terminal state \mathbf{x}_{\perp} is reached.

In **analytical mechanics** L is known as the **Lagrangian**, and the time-integral of L is called the **action**:

$$\boxed{\text{action } S} = \int L dt \quad (24)$$

Hamilton's **principle of least action** says that S always takes the **stationary value**, ie, the S value is extremal compared with neighboring trajectories.

The **Hamiltonian** is defined as $H = L + \frac{\partial J^*}{\partial \mathbf{x}} \mathbf{f}$, which arises from the method of **Lagrange multipliers**.

All these refer to essentially the same thing, so we have the following correspondence:

Reinforcement learning	Control theory	Analytical mechanics
utility or value U	cost J	action S
instantaneous reward R	running cost	Lagrangian L
action a	control u	(external force?)

 (25)

Interestingly, the reward R corresponds to the **Lagrangian** in physics, whose unit is “energy”; In other words, “desires” or “happiness” appear to be measured by units of “energy”, this coincides with the idea of “positive energy” in pop psychology. Whereas, long-term value is measured in units of [energy \times time].

This correspondence between these 3 theories is explained in detail in Daniel Liberzon's book [?]. The traditional AI system is discrete-time; converting it to continuous-time seems to

increase the computational burden. The recent advent of **symplectic integrators** [?] are known to produce better numerical solutions that retain qualitative features of the exact solution, eg. quasi-periodicity.

An interesting insight from control theory is that our system is a Hamiltonian dynamical system in a broad sense.

Hamilton's **principle of least action** says that the trajectories of dynamical systems occurring in nature always choose to have their action S taking **stationary values** when compared to neighboring paths. The action is the time integral of the Lagrangian L :

$$\boxed{\text{Action } S} = \int \boxed{\text{Lagrangian } L} dt \quad (26)$$

From this we see that the Lagrangian corresponds to the instantaneous “rewards” of our system. It is perhaps not a coincidence that the Lagrangian has units of **energy**, in accordance with the folk psychology notion of “positive energy” when we talk about desirable things.

The **Hamiltonian** H arises when we consider a typical control theory problem; The system is defined via:

$$\text{state equation:} \quad \dot{\mathbf{x}}(t) = \mathbf{f}[\mathbf{x}(t), \mathbf{u}(t), t] \quad (27)$$

$$\text{boundary condition:} \quad \mathbf{x}(t_0) = \mathbf{x}_0, \mathbf{x}(t_\perp) = \mathbf{x}_\perp \quad (28)$$

$$\text{objective function:} \quad J = \int_{t_0}^{t_\perp} L[\mathbf{x}(t), \mathbf{u}(t), t] dt \quad (29)$$

The goal is to find the optimal control $\mathbf{u}^*(t)$.

Now apply the technique of **Lagrange multipliers** for finding the maximum of a function, this leads to the new objective function:

$$U = \int_{t_0}^{t_\perp} \{L + \boldsymbol{\lambda}^T(t) [f(\mathbf{x}, \mathbf{u}, t) - \dot{\mathbf{x}}]\} dt \quad (30)$$

So we can introduce a new scalar function H , ie the Hamiltonian:

$$H(\mathbf{x}, \mathbf{u}, t) = L(\mathbf{x}, \mathbf{u}, t) + \boldsymbol{\lambda}^T(t) f(\mathbf{x}, \mathbf{u}, t) \quad (31)$$

Physically, the unit of \mathbf{f} is velocity, while the unit of L is energy, therefore $\boldsymbol{\lambda}$ should have the unit of **momentum**. This is the reason why the phase space is made up of the diad of (position, momentum).

According to control theory, the **optimal path** is given by the Hamilton-Jacobi-Bellman equation:

$$\boxed{\text{Hamilton-Jacobi-Bellman}} \quad 0 = \frac{\partial S^*}{\partial t} + \min_u H. \quad (32)$$

With the substitution $\Psi = e^{iS/\hbar}$ into the Hamilton-Jacobi equation, one can obtain the **Schrödinger equation** in quantum mechanics:

$$\boxed{\text{Hamilton-Jacobi}} \quad \frac{\partial S}{\partial t} = -H \quad \xrightarrow{\Psi = \exp\{iS/\hbar\}} \quad i\hbar \frac{\partial \Psi}{\partial t} = H\Psi \quad \boxed{\text{Schrödinger}} \quad (33)$$

which suggests that techniques in quantum mechanics can be applied to solve our AGI problem.

1.7 Prior art: other cognitive architectures

The minimalist architecture based on reinforcement learning has been proposed by Itimar Ariel from Israel, in 2012 [?], and I also independently proposed in 2016 (precursor of this paper). The prestigious researcher of signal processing, Simon Haykin, recently also used the “RL + memory” design, cf. his 2012 book *Cognitive dynamic systems* [?]. Vladimir Anashin in the 1990’s also proposed this kind of cognitive architecture [?]. There may exist more precedents, eg: [?].

2 Logic structure

The transition function \mathbf{F} appearing in (6) is “free” without further restrictions. The learning of \mathbf{F} may be slow without further **induction bias**, cf the “no free lunch” theorem. But we know that the transition function is analogous to \vdash , the logic consequence or entailment operator. So we want to impose this logic structure on \mathbf{F} .

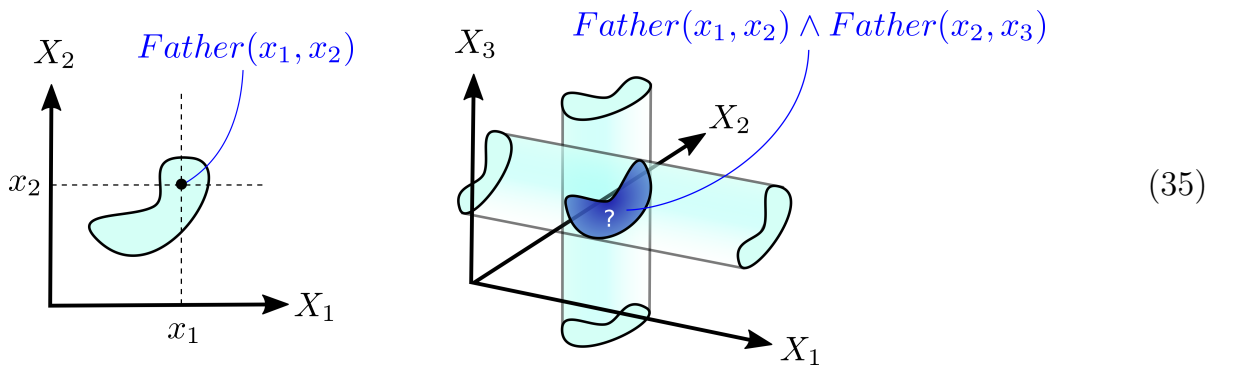
By logic structure we mean that \mathbf{F} would act like a knowledge base $\boxed{\text{KB}}$ containing a large number of logic **rules**, as in the setting of classical logic-based AI.

A logic rule is a conditional formula with variables. For example:

$$\forall X \forall Y \forall Z. \text{ father}(X, Y) \wedge \text{ father}(Y, Z) \Rightarrow \text{ grandfather}(X, Z) \quad (34)$$

where the red lines show what I call “linkages” between different appearances of the same variables.

Quantification of logic variables, with their linkages, result in **cylindrical** and **diagonal** structures when the logic is interpreted *geometrically*. This is the reason why Tarski discovered the **cylindric algebra** structure of first-order predicate logic. That cylindrical shapes can arise from quantification is illustrated below:



And “linkages” cause the graph of the \vdash map to *pass through* diagonal lines such as follows:



We are trying to use neural networks to approximate such functions (*ie*, these geometric shapes). Since NNs are universal function approximators, they can in principle achieve that. There is also *empirical* evidence that NNs can well-approximate logical maps, because the *symbolic* matching and substitution of logic variables is very similar to what occurs in *machine translation* between natural languages; And we know that deep learning is fairly successful at the latter task.

So what exactly is the logic structure? Recall that inside our RL model:

- state = mental state = set of logic propositions
- environment = state space = mental space
- actions = logic rules

For our current prototype system, an action = a logic **rule** is of the form:

$$\overbrace{A_1^1 A_2^1 A_3^1 \wedge A_1^2 A_2^2 A_3^2 \wedge \dots \wedge A_1^k A_2^k A_3^k}^{\text{conjunction of } k \text{ literal propositions}} \Rightarrow A_1^0 A_2^0 A_3^0 \quad (37)$$

each literal made of m atomic concepts, $m = 3$ here

where an atomic concept, or just **atom**, can be roughly understood as a **word vector** as in Word2Vec. Each $A \in \mathbb{R}^d$, where d is the dimension needed to represent a single word vector or atom.

We use a “free” neural network (*ie*, standard feed-forward NN) to approximate the set of *all* rules. The **input** of the NN would be the state vector \mathbf{x} :

$$A_1^1 A_2^1 A_3^1 \wedge A_1^2 A_2^2 A_3^2 \wedge \dots \wedge A_1^k A_2^k A_3^k \quad (38)$$

We fix the number of conjunctions to be k , with the assumption that conjunctions of length $< k$ could be filled with “dummy” (always-true) propositions.

The **output** of the NN would be the conditional **probability** of an action:

$$P(\text{action} \mid \text{state}) := \pi(A_1 A_2 A_3 \mid \mathbf{x}). \quad (39)$$

Note that we don’t just want the action itself, we need the **probability distribution** over these actions. The **Bellman update** of reinforcement learning should update the probability distribution over such actions.

3 Implementation issues

3.1 Commutative / symmetric neural networks

The logic conjunction \wedge is **commutative**:

$$\mathbf{p} \wedge \mathbf{q} \Leftrightarrow \mathbf{q} \wedge \mathbf{p}. \quad (40)$$

If we want to use a neural network to model the deduction operator $\vdash: \mathbb{P}^k \rightarrow \mathbb{P}$, where \mathbb{P} is the space of literal propositions, then this function must be **symmetric** in its input arguments.

A simple fact: If $F(\mathbf{p}, \mathbf{q})$ is any function, then

$$F(\mathbf{p}, \mathbf{q}) + F(\mathbf{q}, \mathbf{p}) \quad \text{or} \quad F(\mathbf{p}, \mathbf{q}) \cdot F(\mathbf{q}, \mathbf{p}) \quad (41)$$

would be symmetric functions in (\mathbf{p}, \mathbf{q}) . This can be easily extended to \mathbb{P}^k . We will use the additive method.

The **back-propagation** algorithm can be easily adapted to symmetric NNs. The gradient of the error, $\nabla \epsilon$, is calculated as usual, in which the key step involves:

$$\nabla F_{\text{sym}}(\mathbf{x}) = \nabla \sum_{\sigma \in \mathfrak{S}_k} F(\sigma \cdot \mathbf{x}) \quad (42)$$

where \mathfrak{S}_k is the symmetric group of k elements, and $\mathbf{x} = \mathbf{p}_1 \wedge \mathbf{p}_2 \wedge \dots \mathbf{p}_k$.

3.2 Probability distribution over continuous actions

All the “knowledge” of the agent is contained in the Q -learning function:

$$\begin{aligned} Q: \mathbb{X} \times \mathbb{A} &\rightarrow [0, 1] \in \mathbb{R} \\ (\mathbf{x}, \mathbf{a}) &\mapsto P(\mathbf{a} \mid \mathbf{x}) \end{aligned} \quad (43)$$

where \mathbb{X} = state space, \mathbb{A} = action space, $P(\cdot)$ = probability distribution.

The function space of Q is equivalent to:

$$\mathbb{X} \rightarrow \mathbb{R}(\mathbb{A}) = \mathbb{R}^{\mathbb{A}} \quad (44)$$

which is very large. For example, if \mathbb{A} has finitely 10 discrete actions, $\mathbb{R}(\mathbb{A})$ would be \mathbb{R}^{10} . It would be much worse if \mathbb{A} is continuously-valued, but there exists a number of techniques to deal with continuous actions in the RL literature.

For our purpose, I think the **Gaussian kernel** (*ie*, radial basis function) method would be very effective. The true probability distribution is approximated by:

$$P(\mathbf{a}) \approx \hat{P}(\mathbf{a}) := \frac{1}{Nh} \sum_{i=1}^N \Phi\left(\frac{\mathbf{a} - \mathbf{a}_i}{h}\right) \quad (45)$$

where $\Phi(u)$ denotes the Gaussian kernel $= \frac{1}{\sqrt{2\pi}} e^{-u^2/2}$.

For each state x , our NN should output a probabilistic *choice* of c actions. So we only need to maintain c “peaks” given by Gaussian kernels. Each peak is determined by its mean (a vector) and variance (a scalar). We fix the variance globally as the parameter h . So our NN, *ie*, Q -function, would be of the form:

$$Q: \mathbb{X} \rightarrow \mathbb{A}^c \quad (46)$$

where each $\mathbf{A} \in \mathbb{A}$ is of the form $\mathbf{A}_1 \mathbf{A}_2 \mathbf{A}_3$ and is of size \mathbb{R}^{3d} , as explained above. Thus our NN is of the form:

$$\begin{aligned} Q: \mathbb{X} &\rightarrow \mathbb{A}^c \\ &= (\mathbb{R}^{3d})^c \rightarrow (\mathbb{R}^{3d})^c \\ &= \mathbb{R}^{3dc} \rightarrow \mathbb{R}^{3dc} \end{aligned} \quad (47)$$

3.3 Policy gradient method

In the policy-gradient method, the policy $\pi(\mathbf{a}|\mathbf{x})$ is expressed as a function parametrized by Θ . The policy is updated via the rule:

$$\Theta \stackrel{+}{=} \eta \nabla_{\Theta} J \quad (48)$$

where η is the **learning rate**, J is the objective function, which is the expectation of the total reward R along a trajectory τ :

$$J = \mathbb{E}_{\tau} [R(\tau)]. \quad (49)$$

The gradient of J can be derived to this formula:

$$\nabla_{\Theta} J = \nabla_{\Theta} \mathbb{E}_{\tau} [R(\tau)] = \mathbb{E}_{\tau} [\nabla_{\Theta} \sum_t \log \pi(\mathbf{a}_t | \mathbf{x}_t; \Theta) R(\tau)]. \quad (50)$$

3.4 Algorithm

The algorithm is exactly the same as the standard Q -learning algorithm:

```
Initialize all  $Q(x, a)$  arbitrarily
For all episodes
  Initialize  $x$ 
  Repeat
    Choose  $a$  using policy derived from  $Q$ , eg  $\epsilon$ -greedy
    Take action  $a$ , observe  $R$  and  $x'$ 
    Update  $Q(x, a)$ :
       $Q(x, a) \stackrel{+}{=} \eta [ R + \gamma \max_{a'} Q(x', a') - Q(x, a) ]$ 
     $x \leftarrow x'$ 
  Until  $x$  is terminal state
```

4 Future directions

- **Memory:** In this minimal architecture there is no episodic memory, this will be dealt with in a later version.

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

$$\text{policy} : \text{state} \xrightarrow{\text{action}} \text{state}' \quad (51)$$

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

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

$$\begin{array}{ccc}
 & & K'_1 \\
 & \nearrow \text{RL} & \\
 K & & \\
 & \searrow \text{RNN} & \\
 & & K'_2
 \end{array}
 \begin{array}{c}
 \\
 \approx \\
 \\
 \end{array}
 \tag{52}$$

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

At state K , we estimate the Q-value $Q(K \overset{a}{\mapsto} K')$. The action that would be chosen at state K is $\arg \max_a Q(K \overset{a}{\mapsto} K')$. This could be used to train the RNN via $K \vdash_W \dots^n K'$.

$$\begin{array}{ccccccc}
 \text{RL} & K & Q(K \mapsto K') & \text{RL} & \text{RL} & \text{RNN} & \text{RL} & K \\
 \text{RNN} & & \text{RNN} & \text{RL} & & \text{RNN} & Q &
 \end{array}$$

RNN “ n -fold”

- stochastic forward-backward propagation
- genetic?
- Hebbian learning

$$\begin{array}{cccccccc}
 \text{RNN} & & \text{recurrent hetero-associative memory} & & \text{input} & \text{Word2vec} & \text{encoding} & \\
 \text{encoding} & & K & \text{RL} & Q & & & \\
 & & \text{generalization} & K & & K \mapsto K' & \text{rule} & \text{yes} \\
 & & \text{decision tree} & & & & & \\
 & & \text{generalization} & & \text{generalization} & & & \\
 \text{RNN} & & \text{representation} & & \text{multi-step logic} & \text{forward / back-} & & \\
 \text{ward chaining} & & & & \text{concatenation} & \text{“}n\text{-fold”} & & \\
 \text{RL} & \text{generalization} & \text{rules} & \text{generalization} & K & K & & \\
 \text{RL} & \text{RNN} & & & & & & \\
 \text{heterarchical} & \text{decision tree} & & & \text{objects} & \text{fea-} & & \\
 \text{tures} & \text{mapping} & \text{mapping} & \text{rules} & K & K' & & \\
 & = & \text{iteration} & & K \mapsto K' & \text{mapping} & & \\
 \text{setup} & \text{logic} & \text{combinatorial “feel”} & \text{rules} & & \text{continuous} & &
 \end{array}$$

= single-step rules RL rewards rewards

- RL generalization
- iterative thinking map learn
-

Hebbian I/O pattern strengthen pattern

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

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

$$\begin{array}{ccc}
 & & K'_1 \\
 & \nearrow^a & \vdots \\
 K & & \approx \\
 & \searrow_{D^n} & \vdots \\
 & & K'_2
 \end{array} \tag{53}$$

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 $Q(\text{state}, \text{action})$, usually stored as a *look-up table*, 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.

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.

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

$$K_0 \xrightarrow{a} K_+ \quad \updownarrow \star \quad (54)$$

$\updownarrow \star$ 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

5 Misc points

- If sigmoid is replaced by polynomial, universal approximating property may be retained.
- Banach fixed point theorem does not apply because R in general need not be contractive. Question is whether R necessarily converges to fixed points and the answer is no.
- If reasoning operator R is continuous, the flow of the dynamical system is governed by an autonomous differential equation. Poincare-Bendixson only applies to dynamical systems on the plane, and is irrelevant to systems whose phase space has dimension ≥ 3 , or to discrete dynamical systems.
- Time can be discrete or continuous.
- Goal is to find minimizer of error (ie, to approximate a function given some input-output data points). The (finite) set of local minima can be solved via setting $\frac{\partial R}{\partial W} = 0$. The number of local minima can be calculated as: ? McClelland paper.
- If operator is discontinuous, what advantages can be gained?

What I want to do now is to determine if R implemented as a deep network is sufficient to model human-level reasoning.

One principle seems to be that logical conclusions must not proliferate indefinitely. But we are not sure what kind of structural constraints this would impose on the vector space. Or whether we should impose such constraints manually.

What other properties are desired for the implementation of R ?

6 Architecture

TO-DO: The state space X may be too large and we may need an **attention mechanism** to select some parts of X for processing by R . This is the notion of **working memory** in cognitive science.

7 Deep Recurrent Learning

The learning algorithm for R is central to our system. R learns to recognize input-output pairs (\vec{x}_0, \vec{x}^*) . What makes it special is that R is allowed to iterate a *flexible* number of times before outputting an answer. In feed-forward learning we simply learn single-pass recognition, whereas in common recurrent learning we train against a *fixed* time sequence. Here, the time delay between input and output is allowed to stretch arbitrarily.

Suppose the recurrent network R iterates n times:

$$\vec{x}_{t+1} = \overbrace{R \circ R \circ \dots}^n(\vec{x}) \quad (55)$$

As $n \rightarrow \infty$, we get the continuous-time version (a differential equation):

$$\frac{d\vec{x}(t)}{dt} = \mathfrak{R}(\vec{x}(t)) \quad (56)$$

We could run the network R for a long enough time T such that it is highly likely to reach an equilibrium point. Then:

$$\vec{x}_T = \int_0^T \mathfrak{R}(\vec{x}(t)) dt \quad (57)$$

and the error:

$$\mathcal{E} = \vec{x}^* - \vec{x}_T \quad (58)$$

where \vec{x}^* is the target value which is independent of time.

$$\begin{aligned} \frac{\partial \mathcal{E}}{\partial \vec{W}} &= -\frac{\partial}{\partial \vec{W}} \int_0^T \mathfrak{R}(\vec{x}(t)) dt \\ &= -\frac{\partial}{\partial \vec{W}} \int_0^T \bigcirc(W_1 \bigcirc(W_2 \dots \bigcirc(W_L \vec{x}(t))) dt \end{aligned} \quad (59)$$

When there are many layers or if the recurrence is too long, back-prop learning becomes ineffective due to the **vanishing gradient** problem. One solution is to use the **rectifier** activation function: Since its derivative is piecewise constant, it does not suffer from the vanishing gradient problem.

7.1 Forward-backward Algorithm

This is inspired by forward- and backward-chaining in LBAI. We propagate the state vector from both the initial state \vec{x}_0 as well as the final state \vec{x}^* . This bi-directional propagation is added with noise and repeated many times, thus implementing a **stochastic local search**:

When the forward and backward states get close enough, a successful path is found, and we record the gap and the noises along the path, and use them to train R so that this new path would be recognized.

One key question is how to deal with "don't care" bits? One answer is that their errors are zero. But then this is the same as the error for "correct" weights, which seems not well. There's got to be a way to alter weights when the answer is correct...

For $\# \text{ Iteration} = 0$, output is immediately known, so potentially the training can be done. But how to convey that all these alterations of weights are **optional**?