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).

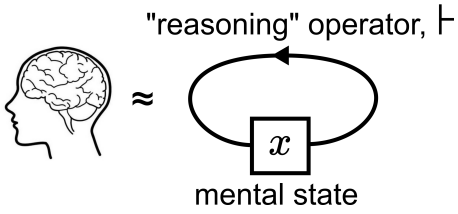
The rest of this paper will explain these design features in detail.

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:

(1)

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



(2)

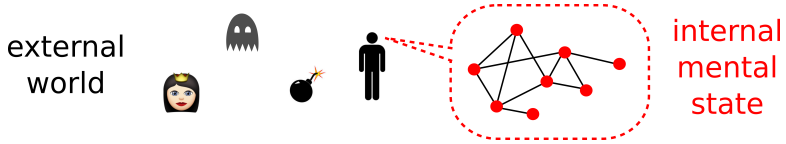
A mental state is a **set of propositions**, for example:

- 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:



(3)

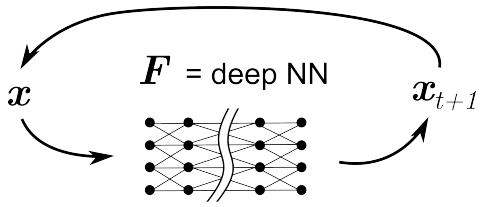
1.2 Actions = cognitive state-transitions = "thinking"

Our system consists of two main algorithms:

1. Learning the transition function $\vdash$ or $\boldsymbol{F} : \boldsymbol{x} \mapsto \boldsymbol{x}'$. $\boldsymbol{F}$ represents the **knowledge** that constrains thinking. In other words, the learning of $\boldsymbol{F}$ is the learning of "static" knowledge.

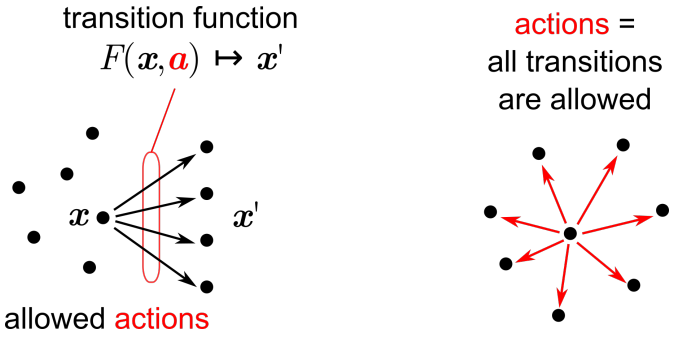
2. Finding the optimal trajectory of the state $\boldsymbol{x}$. This corresponds to optimal "thinking" under the constraints of static knowledge.

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



(4)

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



(5)

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 $\boldsymbol{x} \in \mathbb{X}$ where $\mathbb{X}$ is the space of all possible cognitive states, the reasoning operator $\vdash$ or $\boldsymbol{F}$ is an **endomorphism** (an **iterative map**) $\mathbb{X} \to \mathbb{X}$.

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

$$\boxed{\text{discrete time}} \qquad \boldsymbol{x}_{t+1} = \boldsymbol{F}(\boldsymbol{x}_t) \tag{6}$$

$$\text{or } \boxed{\text{continuous time}} \qquad \dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}) \tag{7}$$

where $\boldsymbol{f}$ and $\boldsymbol{F}$ 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 $\boldsymbol{u}(t)$:

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u}(t), t) \tag{8}$$

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

A typical control-theory problem is described by:

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

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

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

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

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

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

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

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:

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

For details readers may see my *Reinforcement learning tutorial* [8].

U is the total rewards of a sequence of actions:

$$\overbrace{U(\boldsymbol{x}_0)}^{\text{total value of state 0}} = \sum_t \overbrace{R(\boldsymbol{x}_t, \boldsymbol{u}_t)}^{\text{reward at time } t} \tag{15}$$

[1] They are related by: $\boldsymbol{x}(t+1) = \boldsymbol{F}(\boldsymbol{x}(t))$, $\boldsymbol{x}^{-1}(\boldsymbol{x}(t)) = t = \int_{\boldsymbol{x}_0}^{\boldsymbol{x}_t} \frac{d\boldsymbol{x}}{\boldsymbol{f}(\boldsymbol{x}(t))}$, and $f(\boldsymbol{x}) = \frac{1}{(\boldsymbol{x}^{-1})'(\boldsymbol{x}(t))}$. So we can just solve the functional equation $\boldsymbol{x}^{-1}(\boldsymbol{F}(\boldsymbol{x})) - \boldsymbol{x}^{-1}(\boldsymbol{x}) = 1$. See [3] §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**:

$$\text{policy}: \quad \text{state} \xmapsto{\text{action}} \text{state'} \tag{16}$$

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

$$\text{reward}: \boxed{\text{state}} \times \boxed{\text{action}} \to \mathbb{R} \tag{17}$$

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."

value of entire path — reward of choosing $\boldsymbol{u}$ at current state — value of rest of path

$$\boxed{\text{Bellman equation}} \quad U^*(\boldsymbol{x}) = \max_{\boldsymbol{u}}\{\ R(\boldsymbol{u}) + U^*(\boldsymbol{x}_{t+1})\ \} \tag{18}$$

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 $\boldsymbol{u}$ in state $\boldsymbol{x}$. The relation between Q and U is:

$$U(\boldsymbol{x}) = \max_{\boldsymbol{u}} Q(\boldsymbol{x}, \boldsymbol{u}) \tag{19}$$

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:

$$\begin{aligned}\boxed{\text{optimal path}} = {} & \text{choose max reward on current path segment} \\ & + \boxed{\text{the rest of optimal path}}\end{aligned} \tag{20}$$

In math notation:

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

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:

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

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

$$\boxed{\text{utility } U} = \int \boxed{\text{reward } R}\, dt \tag{22}$$

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

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

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 $\boldsymbol{x}_\perp$ is reached.

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

$$\boxed{\text{action } S} = \int L dt \tag{24}$$

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 $H = L + \frac{\partial J^*}{\partial \boldsymbol{x}} \boldsymbol{f}$, 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 [7]. The traditional AI system is discrete-time; converting it to continuous-time seems to

increase the computational burden. The recent advent of **symplectic integrators** [6] 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 occuring 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:

$$\boxed{\text{Action S}} = \int \boxed{\text{Lagrangian L}}\ dt \tag{26}$$

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:

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

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

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

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

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

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

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

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

Physically, the unit of $\boldsymbol{f}$ 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:

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

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

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

which suggests that techiques 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 [2], 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* [4]. Vladimir Anashin in the 1990's also proposed this kind of cognitive architecture [1]. There may exist more precedents, eg: [5].

2 Logic structure

The transition function $\boldsymbol{F}$ appearing in (6) is "free" without further restrictions. The learning of $\boldsymbol{F}$ 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 $\boldsymbol{F}$.

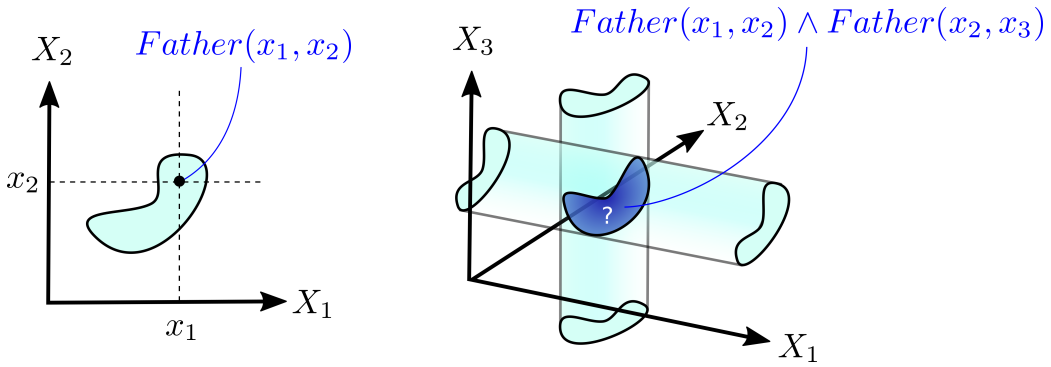
By logic structure we mean that $\boldsymbol{F}$ would act like a knowledge base KB containing a large number of logic **rules**, as in the setting of classical logic-based AI.

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

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

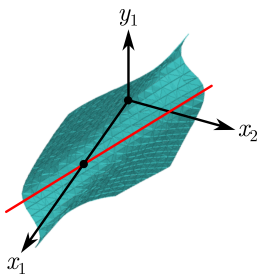
where the red lines show what I call "linkages" between different appearances of the same variables.

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:



(35)

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



(36)

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:

$$\overbrace{\mathsf{A}_1^1\,\mathsf{A}_2^1\,\mathsf{A}_3^1 \;\wedge\; \underbrace{\mathsf{A}_1^2\,\mathsf{A}_2^2\,\mathsf{A}_3^2}_{\text{each literal made of } m \text{ atomic concepts, } m=3 \text{ here}} \wedge \ldots. \wedge\; \mathsf{A}_1^k\,\mathsf{A}_2^k\,\mathsf{A}_3^k}^{\text{conjunction of } k \text{ literal propositions}} \quad \Rightarrow \quad \mathsf{A}_1^0\,\mathsf{A}_2^0\,\mathsf{A}_3^0 \tag{37}$$

where an atomic concept, or just **atom**, can be roughly understood as a **word vector** as in Word2Vec. Each $\mathsf{A} \in \mathbb{R}^d$, 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 $\boldsymbol{x}$:

$$\mathsf{A}_1^1\,\mathsf{A}_2^1\,\mathsf{A}_3^1 \;\wedge\; \mathsf{A}_1^2\,\mathsf{A}_2^2\,\mathsf{A}_3^2 \;\wedge\; \ldots. \;\wedge\; \mathsf{A}_1^k\,\mathsf{A}_2^k\,\mathsf{A}_3^k \tag{38}$$

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:

$$P(\text{action} \mid \text{state}) := \pi(\,\mathsf{A}_1\,\mathsf{A}_2\,\mathsf{A}_3 \mid \boldsymbol{x}). \tag{39}$$

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 conjuction $\wedge$ is **commutative**:

$$\boldsymbol{p} \wedge \boldsymbol{q} \quad \Leftrightarrow \quad \boldsymbol{q} \wedge \boldsymbol{p}. \tag{40}$$

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

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

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

would be symmetric functions in $(\boldsymbol{p}, \boldsymbol{q})$. 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:

$$\nabla \boldsymbol{F}_{\text{sym}}(\boldsymbol{x}) = \nabla \sum_{\sigma \in \mathfrak{S}_k} \boldsymbol{F}(\sigma \cdot \boldsymbol{x}) \tag{42}$$

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

3.2 Probability distribution over continuous actions

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

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

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

The function space of Q is equivalent to:

$$\mathbb{X} \to \mathbb{R}(\mathbb{A}) = \mathbb{R}^{\mathbb{A}} \tag{44}$$

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:

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

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:

$$Q : \mathbb{X} \to \mathbb{A}^c \tag{46}$$

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

$$\begin{aligned} & Q : \mathbb{X} \to \mathbb{A}^c \\ = & (\mathbb{R}^{3d})^k \to (\mathbb{R}^{3d})^c \\ = & \mathbb{R}^{3dk} \to \mathbb{R}^{3dc} \end{aligned} \tag{47}$$

3.3 Policy gradient method

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

$$\Theta \stackrel{+}{=} \eta \, \nabla_\Theta J \tag{48}$$

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

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

The gradient of J can be derived to this formula:

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

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 ε-greedy
        Take action a, observe R and x'
        Update Q(x,a):
            Q(x,a) ± η [R + γ max_a' Q(x',a') − Q(x,a)]
        x ← x'
    Until x is terminal state
```

4 Future directions

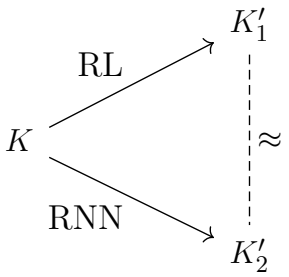
- **Memory:** In this minimal architecture there is no episodic memory, this will be dealt with in a later version.

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

$$\text{policy}: \quad \text{state} \xmapsto{\text{action}} \text{state'} \tag{51}$$

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:



(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 $K \vdash_W ...^n K'$.

RL 在众多状态 K 之间游荡，学习 $Q(K \mapsto K')$。因为 RL 独有奖励讯息，我们必需用 RL 来教导 RNN 学习，反之不可。第一个问题是：RL 如何在 K 之间游荡？游荡是随机的，但也可以借助 RNN 的随机性、或在 RNN 自身的游荡中注入更多随机性、或者根本就是 RL 自己产生的随机性。接下来的问题是：RNN 如何用 Q 值来诱发学习？

RNN 的 "n-fold" 学习可以通过以下方式实现：

- stochastic forward-backward propagation
- genetic?
- 最有趣的是 Hebbian learning，因为它似乎特别适合这情况。

RNN 的本质是什么？它似乎是一个 recurrent hetero-associative memory。但其实它还需要将 input 作类似於 Word2vec 的 encoding。这个 encoding 将「相似」的思维状态 K 归到同类。利用空间中的相似度，RL 可以用一些连续函数来近似 Q 值（详细情况还有待分析）。

另一个问题是：虽然用函数的近似可以做到 generalization，但另一个方法是利用状态 K 中的空位作暂时储存。这两者似乎很不同。问题似乎在於：状态转换 $K \mapsto K'$ 是不是对应於逻辑中的**一条** rule？答案似乎是 yes。这个共识是很重要的。如果用 decision tree，需要的是向量空间中的相似度。

现在的关键是「状态变量」。因为它可以做到符号逻辑中靠变量的 generalization，这是前所未有的。这种 generalization 似乎不需要相似度，因为它是符号的！会不会在向量空间中的状态变量能够做到之前逻辑变量做不到的动作？不管怎样，用 RNN 学习这些变量的动作似乎是很难的，因为这些动作似乎不是对**误差**的梯度下降。除非这些动作本身也近似於其他动作，但那是怎样的近似？学习 multi-step logic 其实和以前的 forward / backward chaining 没有分别！唯一分别是命题的 representation 改变了，它未必像符号的 concatenation。所以问题仍然是 "n-fold" 学习法。

而且注意：RL 的 generalization 根本上不同於 rules 空间中的 generalization。前者是思维空间 K 中的一般化，后者也可以是 K 空间的一般化，但也可以是依赖「状态变量」的一般化。

一般来说，RL 和 RNN 的行动和学习，是可以互相独立的。

还有 heterarchical 的分类法。想用 decision tree 或什么，达到不同网络的**分工**。在组织知识这方面，深度网络有没有用？可以想像，在视觉识别中，在网络的最上层有很多 objects，而它们都可以还原到底层的 features。网络有更多层，可以识别的事物更抽象。但现在我们要的不是**模式识别**，而是 mapping。特别是抽象模式的 mapping。想要的是：大量的 rules，将不同的 K 映射到新的 K'。

还有一点要澄清的是：究竟每一个「思元素」在向量空间中是不是**一点**？如果有了这个「思元素 = 点」假设，则每次 iteration 应该会删除一个思元素，而用另一个（全新的）思元素取代之。这样，$K \mapsto K'$ mapping 就有了更确定的结构。这样的 setup 已经很接近 logic 系统，但其学习算法仍然很有 combinatorial 的 "feel"。（因为只有当两个 rules 串连之后，才能达到某个结论，而这个串连有没有中间的 continuous 状态？）这种串连通常是怎样找到的？

现在有一转机：如果「思元素 = 点」，则「状态变量」的形成似乎会很普遍，而我们可以集中研究如何学习 single-step rules。RL 的 rewards 可以指导学习，但这些「终极 rewards」对学习的细节没有指导作用。我们似乎可以用「**时间延迟**」来达到「状态变量」的效果，这个做法无形中增加了使用状态变量的机会。

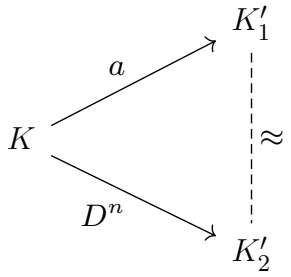
现在总结一下仍然有待回答的问题：

- RL 的 generalization 如何做？
- iterative thinking map 如何 learn？
-

Hebbian 的情况是：有某一 I/O pattern；我想 strengthen 这 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:



(53)

[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.

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.

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

$$K_0 \overset{a}{\longmapsto} K_{\vdash} \quad \updownarrow \bigstar \tag{54}$$

$\updownarrow \bigstar$ 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

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:

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

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

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

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

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

and the error:

$$\mathscr{E} = \vec{x}^* - \vec{x}_T \tag{58}$$

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

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

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 # 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**?

References

1. Vladimir Anashin and Andrei Khrennikov. *Applied algebraic dynamics.* de Gruyter, 2009.

2. Itamar Arel. *Deep reinforcement learning as Foundations for Artificial Intelligence*, chapter 6, pages 89–102. Atlantis Press, 2012.
3. Dolotin and Morozov. *Introduction to non-linear algebra*. World Scientific, 2007.
4. Simon Haykin. *Cognitive dynamic systems*. Cambridge Univ Press, 2012.
5. Vladimir Ivancevic and Tijana Ivancevic. *Geometrical dynamics of complex systems: a unified modeling approach to physics, control, biomechanics, neurodynamics and psycho-socio-economical dynamics*. Springer, 2006.
6. Leimkuhler and Reich. *Simulating Hamiltonian dynamics*. Cambridge University Press, 2009.
7. Daniel Liberzon. *Calculus of variations and optimal control theory: a concise introduction*. Princeton Univ Press, 2012.
8. King Yin Yan. Reinforcement learning tutorial `https://drive.google.com/file/d/0{B}x3_S9{S}Exak-X29uazI1YnhoNFU/view`.