Soar Workshop
RL Tutorial

May 6, 2019

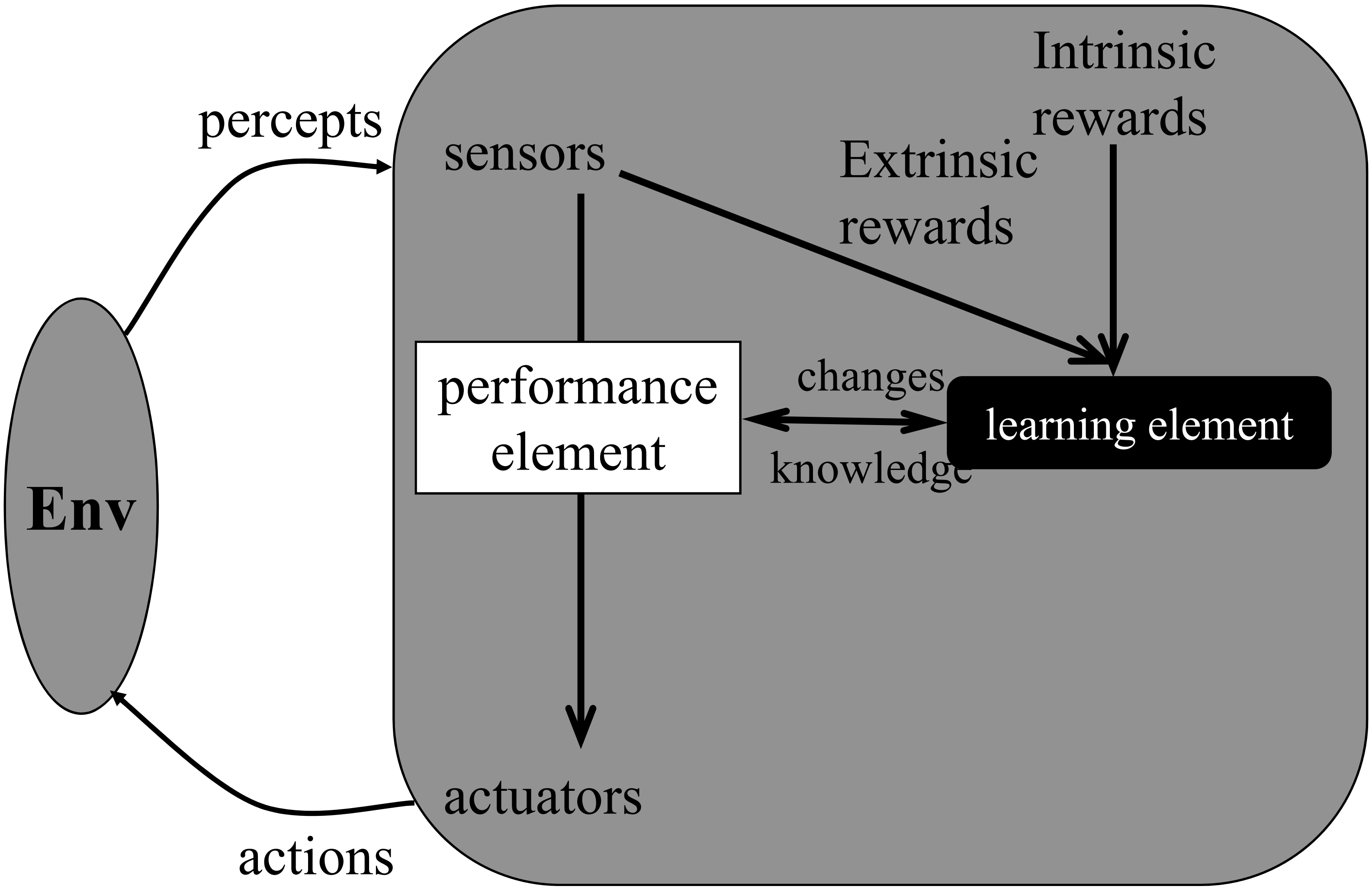
1. RL as a learning mechanism
2. Architecture & agent design
3. Eater integration

Broader Learning Context

From Yann LeCun

- **"Pure" Reinforcement Learning** (cherry)
  - The machine predicts a scalar reward given once in a while.
  - **A few bits for some samples**
- **Supervised Learning** (icing)
  - The machine predicts a category or a few numbers for each input
  - Predicting human-supplied data
  - **10→10,000 bits per sample**
- **Unsupervised/Predictive Learning** (cake)
  - The machine predicts any part of its input for any observed part.
  - Predicts future frames in videos
  - **Millions of bits per sample**

Reinforcement Learning Agent



Reinforcement Learning Agent

- Learns by <u>doing!</u>
- How animals learn by (lots of) training
- Good for optimizing decision making
- Requires frequent reward

- RL makes explicit the *trade-off* between
  - Exploration: acting to *learn* the environment,
  - Exploitation: acting to *maximize reward*.

What Has RL Done?

- Game playing
  - A world-class backgammon player
  - A checkers player in the 1950s
  - Lots of video games in 2015-2016
  - Alpha-Go, Alpha-Go Zero, Alpha-Star in 2016-2019
- Robotics
  - A faster walking algorithm for the Aibo
  - A stable controller for a passive-dynamic walker

Mazes: Demo

What is Reinforcement Learning (RL)?

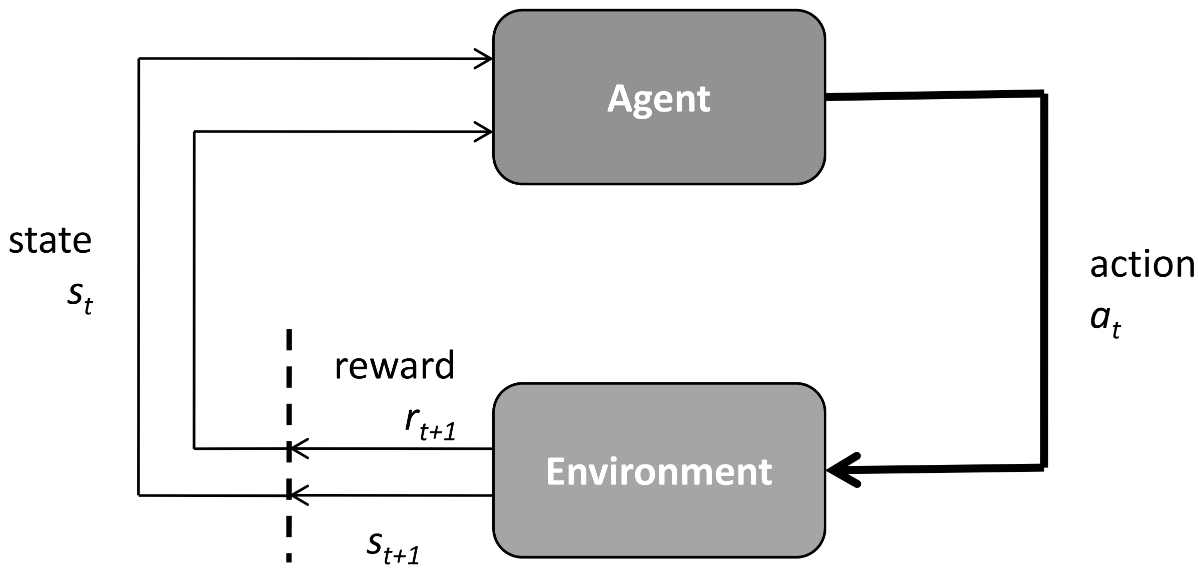
- Goal: learn an optimal action **policy**; given an environment that provides states, affords actions, and provides feedback as numerical **reward**, maximize the expected future reward.
- Typically involves <u>learning</u> a **value function** that maps states (or state-action pairs) to a prediction of expected future reward.

- Allows giving reward for achieving goal and having system figure out how to achieve reward.

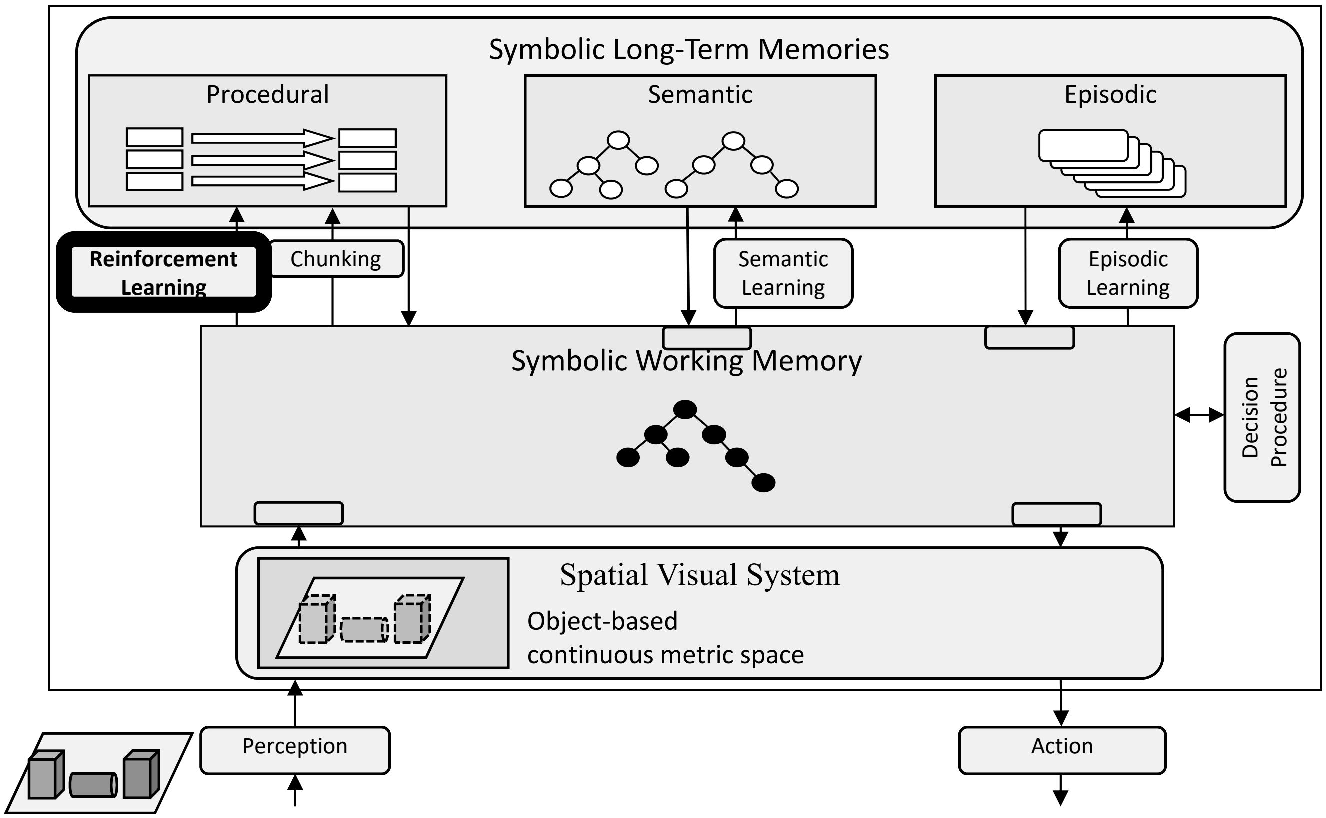
- In Soar, RL involves learning operator *selection* knowledge: numeric preferences.

RL Cycle

Goal: learn an action-selection policy such as to maximize expected receipt of future reward



Soar 9

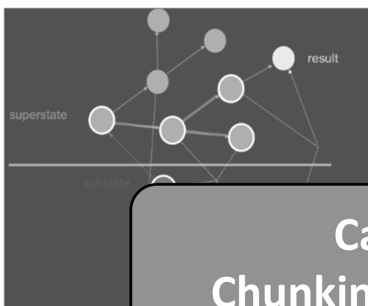


Symbolic Long-Term Memories
Procedural
Semantic
Episodic
Reinforcement Learning
Chunking
Semantic Learning
Episodic Learning
Symbolic Working Memory
Decision Procedure
Spatial Visual System
Object-based
continuous metric space
Perception
Action

Methods for
Learning Procedural Knowledge

Chunking

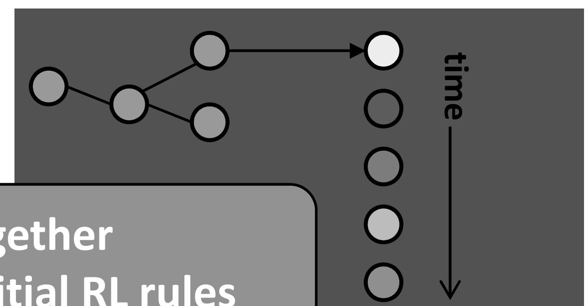
- Converts *deliberation* in substates into *reaction* via rule compilation



- Creates new rules

Reinforcement Learning

- *Tunes* operator numeric preferences to reflect expectation of reward

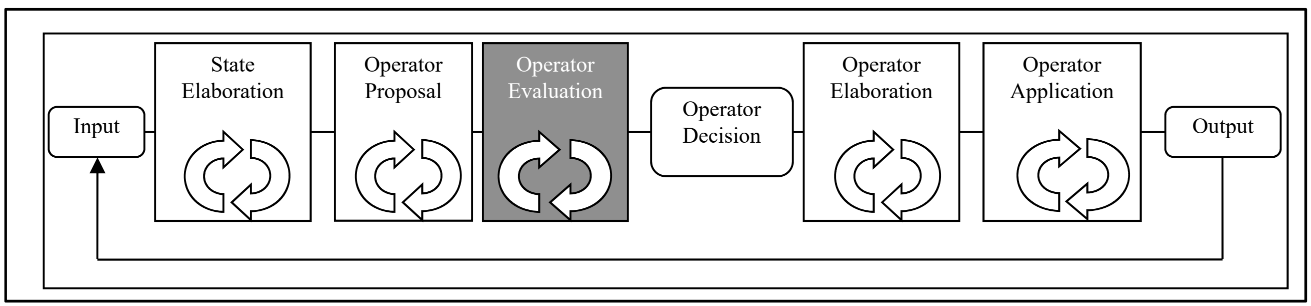


- Updates existing rules

Can be used together
Chunking can learn initial RL rules

Soar Basic Functions

1. <u>Input</u> from environment
2. Elaborate current situation: *parallel rules*
3. Propose operators via acceptable preferences
4. Evaluate operators via *preferences: Numeric indifferent preference*
5. <u>Select operator</u>
6. Apply operator: Modify internal data structures: *parallel rules*
7. <u>Output</u> to motor system [and access to long-term memories]



Left-Right Demo

1. Soar Java Debugger
2. Source `left-right.soar` file

Left-Right Demo

Script

1. `srand 50412`
2. `step`
3. `step`
4. click: `op_pref` tab
   - note numeric indifferent preferences
5. `print left-right*rl*left`
6. `print left-right*rl*right`
7. `step`
   - note movement direction
8. `print left-right*rl*left`
9. `print left-right*rl*right`
10. `init-soar`
11. Repeat from #2 (~5 times)

Left-Right: Takeaways

Reinforcement learning changes rules in procedural memory

- Changes are persistent (until changed by new update).
- Change affects *numeric indifferent preferences*, which in turn affects the selection of operators.
- Change is in the direction of the underlying reward signal (will discuss this more shortly)

RL -> Architecture & Agent Design

Value function

*via RL rules **[agent]***

Reward

*via working-memory structures **[architecture, agent]***

Policy updates

*via Temporal Difference (TD) Learning **[architecture]***

RL Rules

The RL mechanism maintains Q-values for state-operator pairs in operator selection rules, identified by syntax

- Action has a <u>single action</u>, that is a <u>single</u> <u>numeric indifferent preference</u> with a <u>constant value</u>

```
sp {left-right*rl*left
   (state <s> ^name left-right
              ^operator <op> +)
   (<op> ^name move
         ^dir left)
-->
   (<s> ^operator <op> = 0)}
```

```
sp {left-right*rl*right
   (state <s> ^name left-right
              ^operator <op> +)
   (<op> ^name move
         ^dir right)
-->
   (<s> ^operator <op> = 0)}
```

Left-Right Demo

Focus: RL Rules

1. Soar Java Debugger
2. Source `left-right.soar` file
3. `print --full --rl`
4. `run`
5. `print --full --rl`
6. `print --rl`

Reward Representation

Each state in working memory has a `reward-link` structure

Reward is recognized by syntax

```
(<s> ^reward-link <r-link>)
(<r-link> ^reward <r>)
(<r> ^value [integer or float])
```

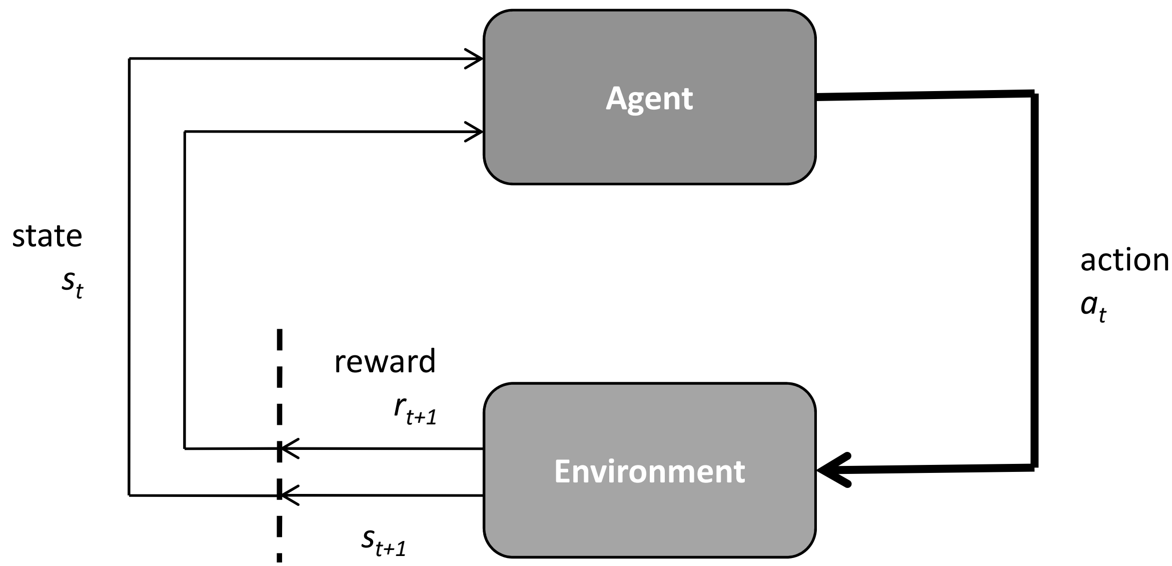
- The reward-link is **not** directly modified by the environment or architecture
- Reward is collected at the beginning of each *decide* phase
- Reward on a state's reward-link pertains only to that state (more on this later)
- Reward can come from multiple rules: reward values are summed by default

Reward Rule Examples

```
sp {left-right*reward*left
   (state <s> ^name left-right
              ^location [redacted]
              ^reward-link <rl>)
-->
   (<rl> ^reward <r>)
   (<r> ^value [redacted]) }
```

```
sp {left-right*reward*right
   (state <s> ^name left-right
              ^location [redacted]
              ^reward-link <rl>)
-->
   (<rl> ^reward <r>)
   (<r> ^value [redacted] }
```

RL Cycle



RL Cycle in Soar

	Input	Propose	Decide	Apply	Output
d					
d+1					

RL Cycle in Soar

	Input	Propose	Decide	Apply	Output
d	$state_d$				
d+1					

RL Cycle in Soar

	Input	Propose	Decide	Apply	Output
d	state_d	evaluate operators_d			
d+1					

RL Cycle in Soar

	Input	Propose	Decide	Apply	Output
d	$state_d$	evaluate $operators_d$	select $operator_d$		
d+1					

RL Cycle in Soar

	Input	Propose	Decide	Apply	Output
d	state_d	evaluate operators_d	select operator_d		initiate external action(s)
d+1					

RL Cycle in Soar

	Input	Propose	Decide	Apply	Output
d	state_d	evaluate operators_d	select operator_d		initiate external action(s)
d+1	state_{d+1} reward_{d+1}				

RL Cycle in Soar

	Input	Propose	Decide	Apply	Output
d	$state_d$	evaluate $operators_d$	select $operator_d$		initiate external action(s)
d+1	$state_{d+1}$ $reward_{d+1}$	evaluate $operators_{d+1}$			

RL Cycle in Soar

	Input	Propose	Decide	Apply	Output
d	state_d	evaluate operators_d	select operator_d		initiate external action(s)
d+1	state_{d+1} reward_{d+1}	evaluate operators_{d+1}	select operator_{d+1} update policy_d		

RL Updates

- Takes place during *decide* phase, after operator selection
- For all RL rule instantiations (**n**) that supported the *last selected* operator

$$value_{d+1} = value_d + (\delta_d / \mathbf{n})$$

Where, roughly…

$$\delta_d = \alpha[reward_{d+1} + \Upsilon(q_{d+1}) - value_d]$$

Where…

- α is a parameter (learning rate)
- Υ is a parameter (discount rate)
- q_{d+1} is dictated by learning policy
  - On-policy (SARSA): value of selected operator
  - Off-policy (Q-learning): value of operator with maximum selection probability

Value Function

Issues

<u>Structure</u>

- What features comprise RL-rule conditions (tradeoff: convergence speed vs. performance)
- Lots of features -> computationally infeasible
- Few features -> not specific enough

<u>Initialization</u>

- Quality estimates may bootstrap agent performance and reduce time to convergence
- Set initial values of the RL rules.

Eaters RL

- General idea:
  - RL rules will learn to select between forward and rotate operators.

Eaters RL 1

Get your eater code

Add to top of file or

create a new file (eater-RL.soar)

– turn on RL

- **`rl -s learning on`**
- **`Indifferent-selection -b`** # use boltzman decision making

Eaters RL 2

Remove indifferent preference from proposals so RL rules will influence decision.

```
sp {random*propose*forward
   (state <s> ^name eater
              ^io.input-link.front)
-->
   (<s> ^operator <op> +)
   (<op> ^name forward)}

sp {random*propose*rotate
   (state <s> ^name eater
              ^io.input-link.front)
-->
   (<s> ^operator <op> +)
   (<op> ^name rotate)}
```

Just add these to a new file and they will load over your old versions.

Eaters RL 3

Generate RL rules for every color and operator combination:

```
gp {eater*evaluate*forward
   (state <s> ^name eater
              ^io.input-link.front [ red wall blue empty green purple ]
              ^operator <op1> +)
   (<op1> ^name forward)
-->
   (<s> ^operator <op1> = 0.0)}

gp {eater*evaluate*rotate
   (state <s> ^name eater
              ^io.input-link.front [ red wall blue empty green purple ]
              ^operator <op1> +)
   (<op1> ^name rotate)
-->
   (<s> ^operator <op1> = 0.0)}
```

Each of these will generate 6 rules!

RL will change the value of = 0.0 in each of the rules as it learns

Eaters RL 4

Add rule that assigns reward – use the change in score:

```
sp {eater*elaborate*state
   (state <s> ^name eater
              ^reward-link <rl>
              ^io.input-link.score-diff <d>)
-->
   (<rl> ^reward.value <d>)
}
```

Run!

- Run eater
- Look at rl rules: `p -r`
- Reset eater (type “r”), run again
- See how rl rules change:
  - Number of updates
  - Value of indifferent preference

- Gets better, but is very limited by the operators available (forward and rotate).