Reinforcement Learning in Soar

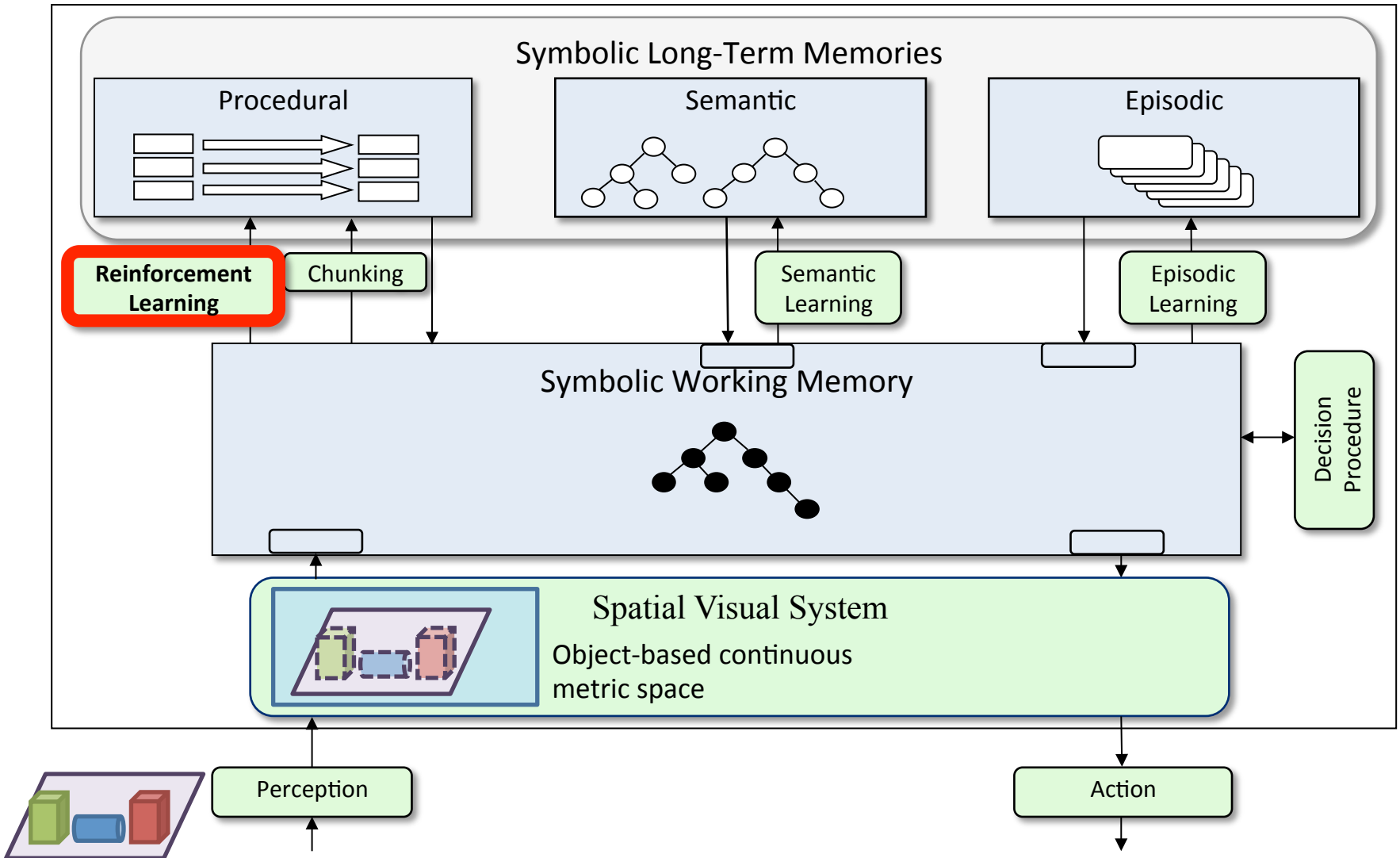
Nate Derbinsky

Wentworth Institute of Technology

Topics

- RL as a learning mechanism
- Architecture & agent design
- Value function structure & initialization

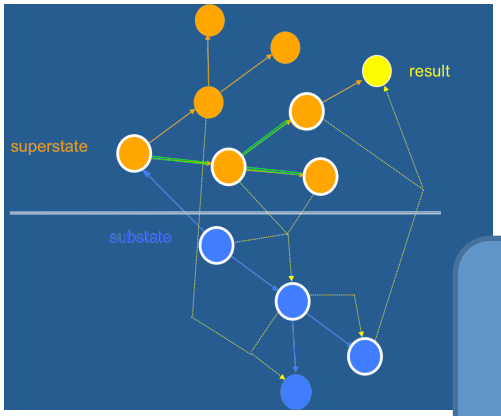
Soar 9


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

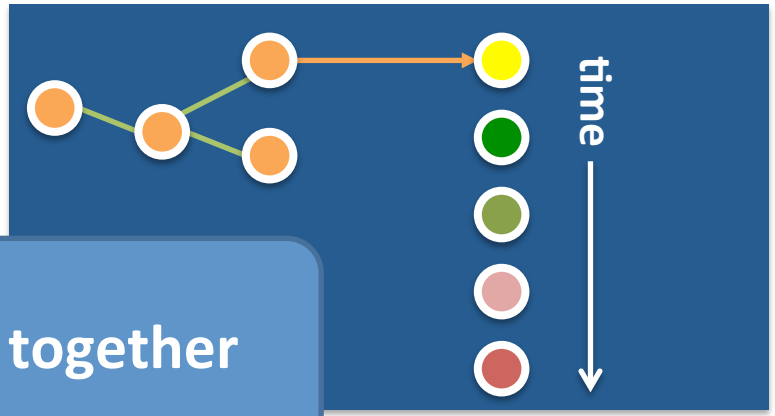
- Converts *deliberation* in substates into *reaction* via rule compilation



- Creates new rules

Reinforcement Learning

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

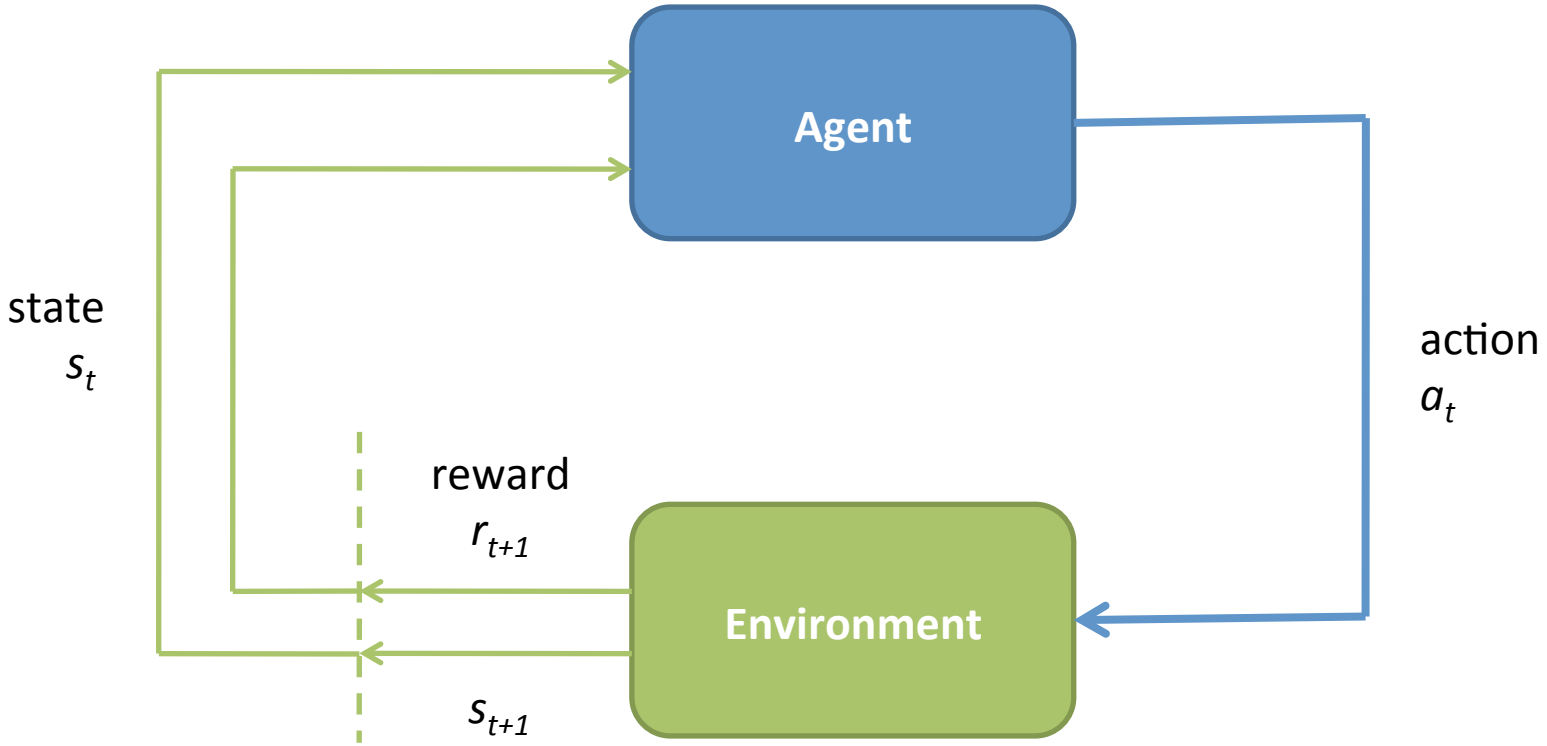


- Updates existing rules

Can be used together

RL Cycle

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



Left-Right Demo

1. Soar Java Debugger
2. Source left-right agent

   `Agents/left-right/left-right.soar`

Left-Right Demo

Script

1. `srand 5041231`
2. `step`
3. `run 1 -p`
4. click: `op_pref` tab
   - note numeric indifferents
5. `print left-right*rl*left`
6. `print left-right*rl*right`
7. `run`
   - 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
- 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 specially formulated rules, identified by syntax

- RHS with a single action, asserting a single numeric indifferent preference with a constant value

```
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 agent

   `Agents/left-right/left-right.soar`

3. `print --full --rl`
4. `run`
5. `print --full --rl`
6. `print --rl`

Reward Representation

Each state in WM has a `reward-link` structure

Reward is recognized by syntax

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

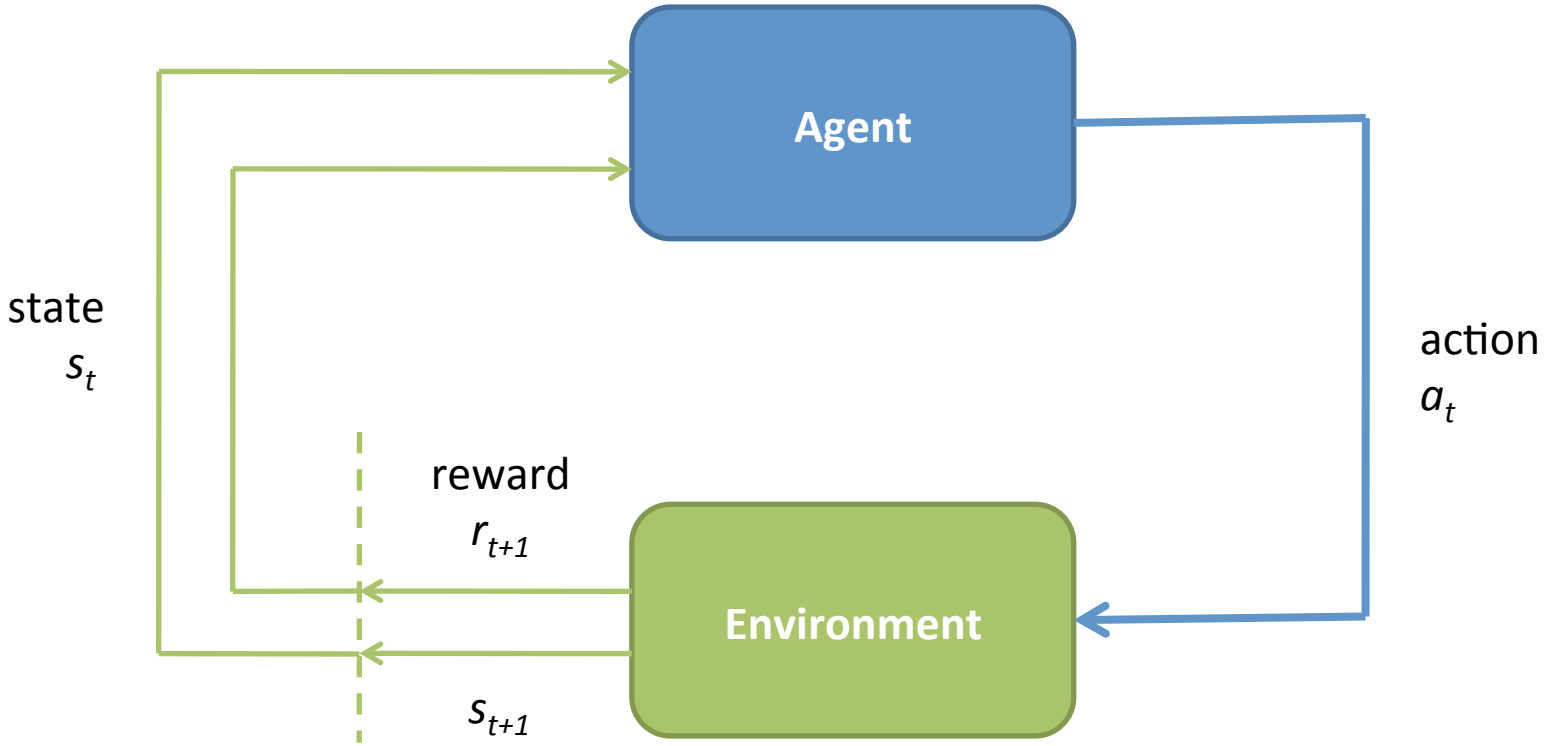
- The reward-link is **not** directly modified by the environment or architecture (i.e. requires agent interpretation/management)
- Reward is collected at the beginning of each *decide* phase
- Reward on a state's reward-link pertains only to that state (more on this later)
- Reward can come from multiple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

Reward Rule Examples

```
sp {left-right*reward*left
   (state <s> ^name left-right
              ^location left
              ^reward-link <rl>)
-->
   (<rl> ^reward <r>)
   (<r> ^value -1)
}
```

```
sp {left-right*reward*right
   (state <s> ^name left-right
              ^location right
              ^reward-link <rl>)
-->
   (<rl> ^reward <r>)
   (<r> ^value 1)
}
```

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

Structure

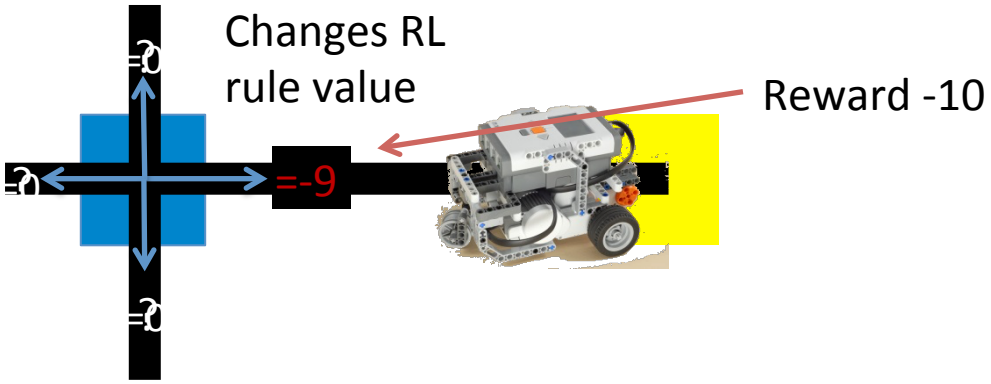
- What features comprise RL-rule conditions (tradeoff: convergence speed vs. performance)
- High dimensionality -> computationally infeasible

Initialization

- Quality estimates may bootstrap agent performance and reduce time to convergence

RL with Mindstorms

- General idea:
  - RL rules will learn to select operators that don't move toward a yellow junction.
  - So, instead of representing a mapping between junction color and direction in working memory, that information is represented in RL rules.



Mindstorms RL 1

Get your Advanced-Stage1 code

Add to top of file – turn on RL

- **`rl -s learning on`**
- **`indiff -g`** # use greedy decision making
- **`indiff -e 0.001`** # low epsilon

Mindstorms RL 2

Remove indifferent preference so RL rules will influence decision.

```
sp {Stage1*propose*select-direction
   (state <s> ^name line-follower
              ^junction-color <color>
             -^selected-direction
              ^direction <dir>)
-->
   (<s> ^operator <o> +)
   (<o> ^name select-direction
        ^direction <dir>)
}
```

Mindstorms RL 3

Generate RL rules for every junction-color and movement combination:

[green red blue brown] X [north south east west]

```
gp {Stage1*propose*select-direction*RL
   (state <s> ^name line-follower
              ^operator <o> +
              ^junction-color [ green red blue brown ]
             -^selected-direction)
    (<o> ^name select-direction
              ^direction [ north south east west ])
-->
   (<s> ^operator <o> = 0)}
```

This generates 16 rules!

RL will change the value of = 0 in each of the rules as it learns

Mindstorms RL 4

Create a negative reward for when the Bot encounters a yellow junction

```
sp {Stage1*evaluate*direction*yellow*bad
    (state <s> ^reward-link <r>
               ^junction-color yellow)
    -->
    (<r> ^reward.value -10)}
```

Comparison to ACT-R

	New Rules	Existing Rules
ACT-R	Compilation • Combines rules that fire in sequence • Affects model timing	Utility • Conflict resolution: Each rule has utility value (U); if multiple match, the highest U fires • Reward via programmatic hooks, rules (learn?) • "Flat" learning via T-1
Soar	Chunking • Generalizes justifications of result(s) from substate reasoning (potentially many decisions/rule firings) • Deliberation -> reaction (potentially fewer decisions, different reasoning paths)	Reinforcement Learning • Numeric indifferent preferences may impact operator selection (resulting in potentially numerous application rules firing) • Reward via rule(s) that can be learned • Hierarchical TD learning

Thank You :)

Questions?