14_lexical_decision_model_production_rules

April 12, 2021

Open in Colab

Model up to this point:

[1]:
```
import pyactr as actr

environment = actr.Environment(focus_position=(0,0))
lex_decision = actr.ACTRModel(
    environment=environment,
    automatic_visual_search=False,
    motor_prepared=True
)
```

[2]:
```
actr.chunktype("goal", "state")
actr.chunktype("word", "form")

dm = lex_decision.decmem

for string in {"elephant", "dog", "crocodile"}:
    dm.add(actr.makechunk(typename="word", form=string))

g = lex_decision.goal

g.add(actr.makechunk(nameofchunk="beginning",
                     typename="goal",
                     state="start"))
```

0.1 The lexical decision model: productions

We only need five productions to model our lexical decision task.

- the first rule requires the visual *Where* buffer to search the (virtual) screen and find the closest word relative to the starting $(0,0)$ position

[3]:
```
lex_decision.productionstring(name="find word", string="""
    =g>
    isa     goal
    state   start
    ?visual_location>
    buffer  empty
    ==>
    =g>
    isa     goal
    state   attend
    +visual_location>
    isa _visuallocation
    screen_x closest
""")
```

```
[3]: {'=g': goal(state= start), '?visual_location': {'buffer': 'empty'}}
     ==>
     {'=g': goal(state= attend), '+visual_location': _visuallocation(color= ,
     screen_x= closest, screen_y= , value= )}
```

The rule requires:

- the `start` chunk to be in the goal buffer (lines 2-4 above)
- the visual location buffer to be empty (lines 5-6)

If these preconditions are met:

- we enter a new goal state of 'attending' to the visual input (lines 8-10)
- the visual location buffer will search for and be updated with the position of the closest element (lines 11-13)
  - the search is launched by specifying `+visual_location`, that is, the name of the buffer and the task +
  - we used + before for the retrieval buffer when we placed a new retrieval request, i.e., we launched a new search in declarative memory
  - we can think of + in the visual *Where* buffer as specifying the same action as in the retrieval buffer, the only difference being that
    * the visual *Where* buffer searches the environment
    * the retrieval buffer searches the declarative memory

Once this rule fires, our ACT-R model will know the position of the closest element on the screen, but it won't know which element is actually present at that location.

To access the element, we make use of the visual *What* buffer, as shown in the `"attend word"` rule below:

```
[4]: lex_decision.productionstring(name="attend word", string="""
    =g>
    isa     goal
    state   attend
    =visual_location>
    isa    _visuallocation
    ?visual>
    state   free
    ==>
    =g>
    isa     goal
    state   retrieving
    +visual>
    isa     _visual
    cmd     move_attention
    screen_pos =visual_location
    ~visual_location>
""")
```

[4]:
```
{'=g': goal(state= attend), '=visual_location': _visuallocation(color= ,
screen_x= , screen_y= , value= ), '?visual': {'state': 'free'}}
==>
{'=g': goal(state= retrieving), '+visual': _visual(cmd= move_attention, color= ,
screen_pos= =visual_location, value= ), '~visual_location': None}
```

This rule checks that:

- the visual *Where* buffer has stored a location (lines 5-6)
- the visual *What* buffer is free, i.e., it is not carrying out any visual action

If these preconditions are satisfied:

- a new chunk is added to the visual *What* buffer that moves the focus of attention to the current visual location (lines 13-16)
- the attention focus is moved by setting the value of the `cmd` (command) slot to `move_attention`
- the goal enters a `retrieving` state (lines 10-12)
- the visual *Where* buffer (a.k.a. `visual_location`) is cleared (line 17)

The interaction between the two vision buffers simulates a two-step process:

1. noticing an object through the visual location (*Where*) buffer
2. finding what that object is, i.e., attending to the object through the visual (*What*) buffer

The next rule starts the memory retrieval process:

- we take the `value =val` of the chunk stored in the visual (*What*) buffer, which is a string, and check to see if there is a word in our lexicon that has that form
- this retrieval request is actually the core part of our lexical decision model
- the crucial parts of the rule are on lines 7 and 14 below:
  - the character string `=val` of the perceived chunk (line 7) becomes the declarative memory cue placed in the retrieval buffer (line 14)

[5]:
```
lex_decision.productionstring(name="retrieving", string="""
    =g>
    isa     goal
    state   retrieving
    =visual>
    isa     _visual
    value   =val
    ==>
    =g>
    isa     goal
    state   retrieval_done
    +retrieval>
    isa     word
    form    =val
""")
```

[5]:
```
{'=g': goal(state= retrieving), '=visual': _visual(cmd= , color= , screen_pos= ,
value= =val)}
==>
{'=g': goal(state= retrieval_done), '+retrieval': word(form= =val)}
```

The final two rules we need are provided below. They consider the two possible outcomes of the retrieval process:

- a lexeme was retrieved, or
- no lexeme was found with that form

[6]:
```
lex_decision.productionstring(name="lexeme retrieved", string="""
    =g>
    isa     goal
    state   retrieval_done
    ?retrieval>
    buffer  full
    state   free
    ==>
    =g>
    isa     goal
    state   done
    +manual>
    isa     _manual
    cmd     press_key
    key     J
""")
```

[6]:
```
{'=g': goal(state= retrieval_done), '?retrieval': {'buffer': 'full', 'state':
'free'}}
==>
{'=g': goal(state= done), '+manual': _manual(cmd= press_key, key= J)}
```

[7]:
```
lex_decision.productionstring(name="no lexeme found", string="""
    =g>
    isa     goal
    state   retrieval_done
    ?retrieval>
    buffer  empty
    state   error
    ==>
    =g>
    isa     goal
    state   done
    +manual>
    isa     _manual
    cmd     press_key
    key     F
""")
```

[7]:
```
{'=g': goal(state= retrieval_done), '?retrieval': {'buffer': 'empty', 'state':
'error'}}
==>
{'=g': goal(state= done), '+manual': _manual(cmd= press_key, key= F)}
```

The format of the rules should look familiar by now. The only new parts are on lines 12-15 (in both rules above):

- these lines set the motor module in action, which can perform only one action, namely pressing a key
- this is implemented by placing a chunk of a special predefined type `_manual` in the `manual` buffer
- the chunk has two slots:
  - `cmd`: what command should be carried out
  - `key` : what key should be pressed
- the command is the same for both rules (`press_key` on line 14)
- the key to be pressed is different:
  - if a lexeme is found, the ACT-R model simulating a participant presses `'J'` (line 15)
  - otherwise, the model presses `'F'` (also line 15)

[ ]:
```

```