13_intro_to_visual_and_motor_modules

April 12, 2021

Open in Colab

1 Syntax as a Cognitive Process: Left-corner parsing with visual & motor interfaces

In the previous notebooks, we introduced and used several ACT-R modules and buffers:

- the declarative memory module and its retrieval buffer
- the procedural memory module and its goal buffer
- the imaginal buffer

These are core ACT-R modules and buffers, but focusing exclusively on them leads to solipsistic models that do not interact in any way with the environment.

In this notebook, we are going to start changing that and introduce the vision and motor modules, which give us basic ways to be affected by and in turn affect the environment outside the mind.

We will then leverage these input/output interfaces when we build a psycholinguistically realistic left-corner parser for the syntactic component of the linguistic representations we will model in this book.

1.1 The environment in ACT-R: modeling lexical decision tasks

We will introduce ACT-R environments by modeling a simple lexical decision task.

- modeling lexical decision tasks is a good stepping stone towards our goal of providing an end-to-end model of self-paced reading
- by end-to-end, we mean a model of self-paced reading that includes both syntactic and semantic parsing (and therefore lexical retrieval of both syntactic and semantic information), and that importantly also has
  - a suitable vision interface to model the way a human perceives the linguistic input incrementally presented on the screen, and
  - a suitable motor interface to model the way a human self-paced reader interacts with the keyboard.

In lexical decision tasks, participants perceive a string and decide whether that string is a word in their language. We will build an ACT-R model to simulate human behavior in this type of tasks.

- the model will search a (virtual) screen and find a string of characters / word on the screen

- if the word matches the (impoverished) lexicon of the model, the model will press the J key on its (virtual) keyboard; otherwise, it will press the F key

We start by importing `pyactr` and creating an environment.

- the environment is just a (simulated) computer screen, and a pretty basic one at that: only plain text is supported
- but that is enough for our purposes throughout this course

[1]:
```python
import pyactr as actr

environment = actr.Environment(focus_position=(0,0))
```

When the class `Environment` is initialized, we can specify various parameters.

- here, we only specify `focus_position`, which indicates the position the eyes focus on when the simulation starts

Two other parameters are:

- `simulated_screen_size`, which specifies the physical size of the screen we are simulating in cm (default: 50×28 cm)
- `viewing_distance`, which specifies the distance between the simulated participants eyes and the screen (default: 50 cm)

Now that the environment is initialized, we can initialize our ACT-R model:

[2]:
```python
lex_decision = actr.ACTRModel(
    environment=environment,
    automatic_visual_search=False,
    motor_prepared=True
)
```

This initialization is similar to what we used before except we specify environment-related arguments:

- we state what environment the model / mind is interacting with
- we set `automatic_visual_search` to `False` so that the model does not start searching the environment for input unless we specifically ask it to
- we state that the motor module is prepared
  - setting `motor_prepared` to `False` would signal that we believe the model to be in a situation in which it did not use the motor module that controls key presses in the last few moments
  - this would make sense if we tried to model the first item in the experiment
  - but the lexical decision tasks are long and repetitive and so, it is more realistic to assume that participants have their motor module in a ready state
  - setting `motor_prepared` to `True` assumes that there is no preparation phase in key presses; otherwise, the module would need 250 ms before executing any manual action

Other parameters that can be explicitly set when initializing a model can be listed together with their default values by accessing the `MODEL_PARAMETERS` attribute (we will explain the majority of them as we proceed):

```
[3]: lex_decision.MODEL_PARAMETERS
```

```
[3]: {'subsymbolic': False,
      'rule_firing': 0.05,
      'latency_factor': 0.1,
      'latency_exponent': 1.0,
      'decay': 0.5,
      'baselevel_learning': True,
      'optimized_learning': False,
      'instantaneous_noise': 0,
      'retrieval_threshold': 0,
      'buffer_spreading_activation': {},
      'spreading_activation_restricted': False,
      'strength_of_association': 0,
      'association_only_from_chunks': True,
      'partial_matching': False,
      'mismatch_penalty': 1,
      'activation_trace': False,
      'utility_noise': 0,
      'utility_learning': False,
      'utility_alpha': 0.2,
      'motor_prepared': False,
      'strict_harvesting': False,
      'production_compilation': False,
      'automatic_visual_search': True,
      'emma': True,
      'emma_noise': True,
      'emma_landing_site_noise': False,
      'eye_mvt_angle_parameter': 1,
      'eye_mvt_scaling_parameter': 0.01}
```

We can now add the modules we used before:

- since we are simulating a lexical decision task, we will add some words to declarative memory that the model can access and check against the stimuli in the simulated experiment

```
[4]: 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
```

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

We add three words to our declarative memory using a Python `for` loop.

- this way of adding chunks to memory can save a lot of time if we want to add a lot of elements, e.g., a reasonably sized lexicon.

We also add a chunk into the goal buffer that will get our lexical decision simulation started.

1.1.1 The visual module

The visual module allows the ACT-R model to 'see' the environment. This interaction happens via two buffers:

- `visual_location` searches the environment for elements matching its search criteria
- `visual` stores the element found using `visual_location`

The two buffers are sometimes called the visual *Where* and *What* buffers.

The visual *Where* buffer searches the environment (the screen) and outputs the location of an element on the screen that matches some search criteria. Visual search cues have three possible slots:

- `color`
- `screen_x`: the horizontal position on the screen
- `screen_y`: the vertical position on the screen)

The x and y positions can be specified:

- in precise terms, e.g., find an element at location `screen_x 100 screen_y 100`, where the numbers represent pixels
- approximately:
  - a `screen_x <100` cue would search for elements at screen locations at most 100 pixels from the left edge of the screen
  - a `screen_x >100` cue would search for elements on the complementary side of the screen.

Three other values are possible for the `screen_x` and `screen_y` slots:

- `screen_x lowest` searches for the element with the lowest position on the horizontal axis (the element closest to the left edge);
- `screen_x highest` searches for the element with the highest position on the same axis (the element closest to the right edge);
- `screen_x closest` searches for the closest element to the current focus position (the axis is actually ignored in this case).

The same applies to the `screen_y` slot.

The visual *What* buffer stores the element whose location was identified by the *Where* buffer.

- the *What* buffer is therefore accessed after the *Where* buffer, as we will see when we state the production rules for our lexical decision model

The vision module as a whole is an implementation of an EMMA (Eye Movements and Movement of Attention) model (Salvucci, Dario D. 2001. An integrated model of eye movements and visual encoding. *Cognitive Systems Research* 1:201–220), which in turn is a generalization and simplification of the E-Z Reader model (Reichle, Erik D, Alexander Pollatsek, Donald L Fisher, and Keith Rayner. 1998. Toward a model of eye movement control in reading. *Psychological Review* 105:125).

- while the latter model is used for reading, the EMMA model attempts to simulate any visual task, not just reading

See Staub, Adrian. 2011. Word recognition and syntactic attachment in reading: Evidence for a staged architecture. *Journal of Experimental Psychology: General* 140:407–433 for a more recent discussion of these models.

1.1.2 The motor module

The motor module is limited to the simulation of a key press on the keyboard – or typing, if multiple key strokes are chained.

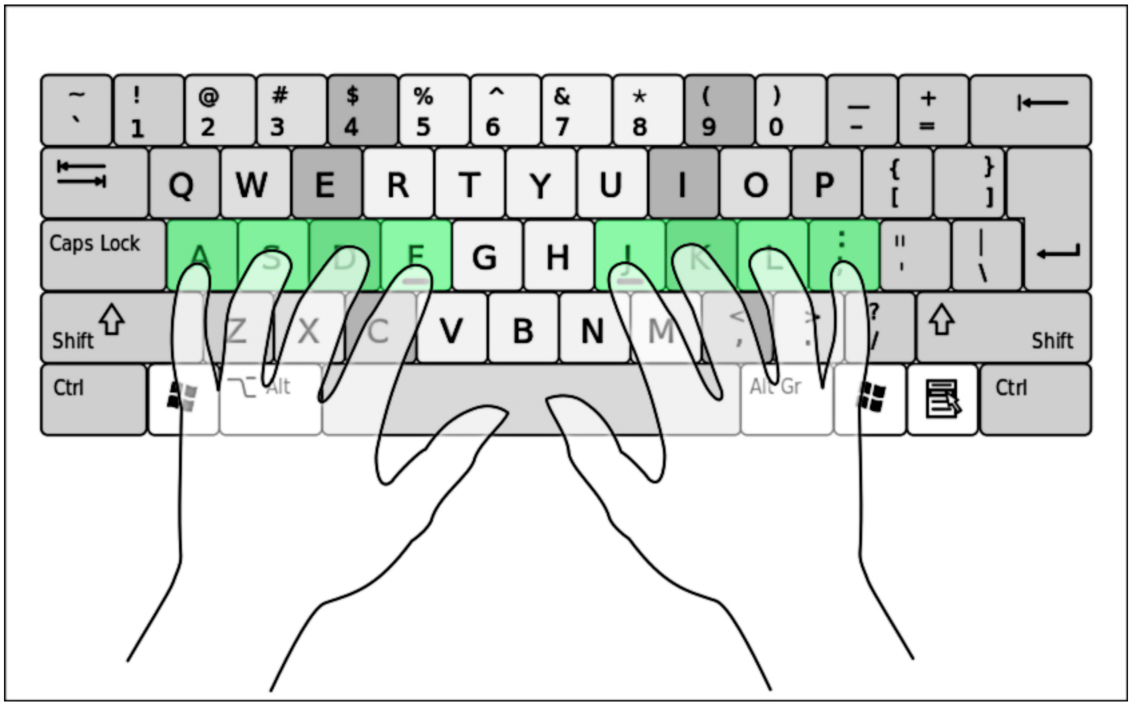
The ACT-R typing model is based on EPIC's Manual Motor Processor (Meyer, David E, and David E Kieras. 1997. A computational theory of executive cognitive processes and multiple-task performance: Part I. Basic mechanisms. *Psychological Review* 104:3).

It has one buffer that accepts requests to execute motor commands.

- the ACT-R motor module currently implemented in `pyactr` is more limited, it currently supports only one command: `press_key`
- this should suffice for simulations of many experimental tasks used in (psycho)linguistics, including lexical decision tasks, self-paced reading, forced-choice tasks etc.
- all of these tasks commonly require only basic keyboard interaction (or mouse button presses, which we will subsume under keyboard interaction) on the participants' part

The hands of the ACT-R model are assumed to be positioned in the home row position on a standard (US) English keyboard, with index fingers at F and J.

- the model assumes a competent, albeit not expert, typist.



(source:Wikipedia)

```
[ ]:
```