2_pyactr_implementation_of_ACTR

April 2, 2021

Open in Colab

1 The `pyactr` implementation of the ACT-R cognitive architecture

1.1 ACT-R implementation

One of the main ways in which this book is different from many other texts in linguistics is its hands-on approach to modeling: we will not only discuss and characterize theoretical claims and language models; we will also implement these models in Python3, making extensive use of the ACT-R package `pyactr`, and we will see what the implemented models predict, down to very specific and fine-grained quantitative details.

The ACT-R theory has been implemented in several programming languages, including Lisp (the 'official' implementation), Java (jACT-R, Java ACT-R), Swift (PRIM) and Python2 (ccm). In this book, we will use a novel Python3 implementation: `pyactr`. This implementation is very close to the official implementation in Lisp, so once you learn it you should be able to fairly easily transfer your newly acquired skills to Lisp ACT-R, if you are so inclined.

However, Python seems to be the *de facto lingua franca* of the scientific computing world:

- it is widely used in the statistics, data science and machine learning communities
- it has a very diverse and robust ecosystem of well-maintained and tested libraries, including an easy-to-use, fast, comprehensive, well-tested and up-to-date scientific computing stack

Because of this, implementing any components that do not directly pertain to ACT-R modeling and the specific linguistic phenomenon under investigation is much easier in Python than in Lisp.

For example, Python makes it much easier to do data manipulation (wrangling/munging) or statistical analysis, to interact with the operating system, to plot results, to incorporate them in an article or book etc. (see `https://xkcd.com/353/`)

Thus, we think `pyactr` is a better tool to learn ACT-R and cognitive modeling:

- the programming language is more familiar and commonly used
- data collection, manipulation, analysis, and presentation, as well as general software maintenance tasks, are much more likely to have good off-the-shelf solutions that require minimal customization

The tool will therefore stand less in the way of the task, so we can focus on actually designing cognitive models, evaluating them and communicating the results.

In addition to the convenience and ease of use that comes with Python, reimplementing ACT-R in `pyactr` also serves to show that ACT-R is a mathematical theory of human cognition that stands on its own, independently of its specific software implementations. While this is well-understood in the cognitive psychology community, it might not be self-evident to working (psycho)linguists or machine-learning researchers.

1.2 Knowledge in ACT-R

There are two types of knowledge in ACT-R:

- declarative knowledge
- procedural knowledge

(see also **Newell, A. 1990.** ***Unified Theories of Cognition*****. Cambridge, MA: Harvard University Press**).

Declarative knowledge is our knowledge of facts. - For example, if one knows what the capital of the Netherlands is, this is encoded and stored in one's declarative knowledge.

Procedural knowledge is knowledge that we display in our behavior (cf. **Newell, A. 1973a. "Production Systems: Models of Control Structures." In** ***Visual Information Processing*****, edited by W.G. Chase and others, 463–526. New York: Academic Press**).

This distinction is closely related to the distinction between explicit knowledge ('knowing that') and implicit knowledge ('knowing how') in analytical philosophy:

- **Ryle, Gilbert. 1949.** ***The Concept of Mind*****. London: Hutchinson's University Library**
- **Polanyi, Michael. 1967.** ***The Tacit Dimension*****. London: Routledge; Kegan Paul**
- a more recent discussion: **Davies, Martin. 2001. "Knowledge (Explicit and Implicit): Philosophical Aspects." In** ***International Encyclopedia of the Social and Behavioral Sciences*****, edited by N. J. Smelser and B. Baltes, 8126–32. Elsevier**

It is often the case that our procedural knowledge is internalized: we are aware that we have it but we would be hard pressed to explicitly and precisely describe it.

Driving, swimming, riding a bicycle and, arguably, using language, are examples of procedural knowledge. Almost all people who can drive, swim, ride a bicycle, talk etc. do so in an 'automatic' manner. They are able to do it but if asked, they might completely fail to describe exactly how they do it.

ACT-R represents these two types of knowledge in two very different ways. Declarative knowledge is encoded in chunks. Procedural knowledge is encoded in production rules, or productions for short.

1.2.1 Declarative memory: chunks

Chunks are lists of attribute-value pairs, familiar to linguists acquainted with feature-based phrase structure grammars (e.g., GPSG, HPSG or LFG). However, in ACT-R, we use the term *slot* instead of *attribute*.

For example, we might think of one's lexical knowledge of the word *car* as a chunk of type WORD with the value 'car' for the slot FORM, the value ⟦car⟧ for the slot MEANING, the value 'noun' for the slot CATEGORY and the value ‘sg’ (singular) for the slot NUMBER. This is represented in graph form below.

$$\begin{tikzcd}
 & \text{sg} & \\
\text{car} & \boxed{car_{\text{WORD}}} \arrow[l, "\text{FORM}"'] \arrow[r, "\text{MEANING}"] \arrow[u, "\text{NUMBER}"'] \arrow[d, "\text{CATEGORY}"] & [\![\text{car}]\!] \\
 & \text{noun} &
\end{tikzcd}$$

The slot values are the primitive elements:

- ‘car’
- ⟦car⟧
- ‘noun’
- ‘sg’

Chunks (complex, non-primitive elements) are boxed and subscripted with their type, whereas primitive elements are simple text. A simple arrow signifies that the chunk at the start of the arrow has the value at the end of the arrow in the slot with the name that labels the arrow.

The graph representation above will be useful when we introduce activations and more generally, ACT-R subsymbolic components in a subsequent notebook. The same chunk can be represented as an attribute-value matrix (AVM). We will primarily use AVM representations like the one below from now on.

$$_{\text{WORD}}\begin{bmatrix} \text{FORM:} & \text{car} \\ \text{MEANING:} & [\![\text{car}]\!] \\ \text{CATEGORY:} & \text{noun} \\ \text{NUMBER:} & \text{sg} \end{bmatrix}$$

1.2.2 Procedural memory: productions

A production is an *if*-statement. It describes an action that takes place if the *if* ‘part’ (the antecedent clause) is satisfied. This is why we think of such productions as ⟨precondition, action⟩ pairs. For example, agreement on a verb can be (abstractly) expressed as follows:

- **If** the number slot of the subject NP in the sentence currently under construction has the value sg (**precondition**),

  then check that the number slot of the main verb also has the value sg (**action**).

Of course, for number agreement in English, this is only half of the story. Another production rule would state a similar ⟨precondition, action⟩ pair for pl number.

Thus, the basic idea behind production rules is that the *if* part specifies preconditions, and if these preconditions are true, the action specified in the *then* part of the rule is triggered.

Having two rules to specify subject-verb agreement might seem like a cumbersome way of capturing agreement that misses an important generalization: the two rules are really just one agreement rule with two distinct values for the number slot. Could we then just state that the verb should have the same number specification as the subject? ACT-R allows us to state just that if we use variables.

A variable is assigned a value in the precondition part of a production, and it has the same value in the action part. In other words, the scope of any variable assignment is the production rule in which that assignment happens.

Given this scope specification for variable assignments, and employing the ACT-R convention that variable names are preceded by =, we can reformulate our agreement rule as follows:

- **If** the number slot of the subject NP in the sentence currently under construction has the value `=x`,

  then check that the number slot of the main verb also has the value `=x`.

1.3 The basics of `pyactr`

We introduce the remainder of the ACT-R architecture by discussing its implementation in `pyactr`. In this section, we describe the inner workings of declarative memory in ACT-R and their implementation in `pyactr`.

In the next section, we turn to a discussion of ACT-R modules and buffers and their implementation in `pyactr`.

We then turn to explaining how procedural knowledge / memory and productions are implemented in `pyactr`.

To use `pyactr`, you have to install it if you're running this notebook in google colab. If you're running it locally on your computer, you might have already installed it.

[1]:
```
# uncomment the line below to install pyactr
# !pip3 install pyactr
```

We then import the relevant package:

[2]:
```
import pyactr as actr
```

We use the `as` keyword so that every time we use methods (functions), classes etc. from the `pyactr` package, we can access them by simply invoking `actr` instead of the longer `pyactr`.

Chunks / feature structures are typed:

- before introducing a specific chunk, we need to specify a chunk type and all the slots / attributes of that chunk type

This is just good housekeeping: by first declaring a type and the attributes associated with that type, we are clear from the start about what kind of objects we take declarative memory to store.

Let's create a chunk type that will encode how our lexical knowledge is stored. We don't strive here for a linguistically realistic theory of lexical representations, we just want to get things off the ground and show the inner workings of ACT-R and `pyactr`:

[3]:
```
actr.chunktype("word", "form, meaning, category, number")
```

The function `chunktype` creates a type `word` with four slots: `form`, `meaning`, `category`, `number`.

- the type name, provided as a character string `"word"`, is the first argument of the function
- the list of slots, with the slots separated by commas, is the second argument

After declaring a type, we can create chunks of that type, e.g., a chunk that will encode our lexical entry for the noun *car*.

[4]:
```
carLexeme = actr.makechunk(nameofchunk="car1",
                           typename="word",
                           form="car",
                           meaning="[[car]]",
                           category="noun",
                           number="sg")
print(carLexeme)
```

```
word(category= noun, form= car, meaning= [[car]], number= sg)
```

The chunk is created using the function `makechunk`, which has two required arguments:

- `nameofchunk`, provided on line 1 above
- `typename` (line 2)

Other than these two arguments (with their corresponding values), the chunk consists of whatever slot-value pairs we need it to contain.

- they are specified as shown on lines 3-6 above.

In general, we do not have to specify the values for all the slots that a chunk of a particular type has; the unspecified slots will be empty.

If you want to inspect a chunk, you can print it, as shown on line 7 above.

Note that the order of the slot-value pairs is different from the one we used when we declared the chunk: for example, we defined `form` first (line 3), but that slot appears second in the output. This is because chunks are unordered lists of slot-value pairs, and Python assumes an alphabetic ordering when printing chunks.

Specifying chunk types is optional. In fact, the information contained in the chunk type is relevant for `pyactr`, but it has no theoretical significance in ACT-R, it is just 'syntactic sugar':

- a chunk type is not identified by the name we choose to give it, but by the slots it has

However, it is recommended to always declare a chunk type before instantiating a chunk of that type. Declaring types:

- clarifies what kind of AVMs are needed in our model
- establishes a correspondence between the phenomena and generalizations we are trying to model, on the one hand, and the computational model itself, on the other hand

For this reason, `pyactr` will print a warning message if we don't specify a chunk type before declaring a chunk of that type. Among other things, this helps us debug our code.

- For example, if we accidentally mistype and declare a chunk of type `"morphreme"` instead of the `"morpheme"` type we previously declared, we would get a warning message that a new chunk type has been created.

It is also recommended that you only use slots already defined in your chunk type declaration (or when you first used a chunk of a particular type).

However, you can always add new slots along the way if you need to: `pyactr` will assume that all the previously declared chunks of the same type had no value for those slots.

For example, imagine we realize half-way through our modeling session that it would be useful to specify what syntactic function a word has. We didn't have that slot in our `carLexeme` chunk.

So let's create a new chunk `carLexeme2`, which is like `carLexeme` except it adds this extra piece of information in the slot `synfunction`. We will assume that the `synfunction` value of `carLexeme2` is `subject`, as shown on line 7 below:

[5]:
```
carLexeme2 = actr.makechunk(nameofchunk="car2",
                            typename="word",
                            form="car",
                            meaning="[[car]]",
                            category="noun",
                            number="sg",
                            synfunction="subject")
print(carLexeme2)
```

```
word(category= noun, form= car, meaning= [[car]], number= sg, synfunction=
subject)
```

```
/usr/local/lib/python3.8/dist-packages/pyactr/chunks.py:130: UserWarning: Chunk
type word is extended with new attributes
  warnings.warn("Chunk type %s is extended with new attributes" % typename)
```

The command goes through successfully, as shown by the fact that we can print `carLexeme2`, but a warning message is issued.

Another, more intuitive way of specifying a chunk uses the method `chunkstring`.

- When declaring chunks with `chunkstring`, the chunk type is provided as the value of the `isa` attribute.
- The rest of the ⟨slot, value⟩ pairs are listed immediately after that, separated by commas.
- A ⟨slot, value⟩ pair is specified by separating the slot and value with a blank space.

[6]:
```
carLexeme3 = actr.chunkstring(string="""
    isa word
    form car
    meaning '[[car]]'
    category noun
    number sg
    synfunction subject""")
print(carLexeme3)
```

```
word(category= noun, form= car, meaning= [[car]], number= sg, synfunction=
subject)
```

The method `chunkstring` provides the same functionality as `makechunk`. The argument `string` defines what the chunk consists of. The slot-value pairs are written as a plain string. Note that we use three quotation marks rather than one to provide the chunk string.

- triple quotation signals that the string can appear on more than one line.

The first slot-value pair, listed on line 2 above, is special. It specifies the type of the chunk, and a special slot is used for this, `isa`. The resulting chunk is identical to the previous one: we print the chunk and the result is the same as before.

- the value of a slot can also be enclosed in quotes, e.g., `'some-value-here'`, i.e., it can be provided as a string; the quotes themselves are not treated as part of the value
- using quotes is needed whenever we want to input non-alphanumeric characters, as we have done when we specified the value of the slot `meaning`.

Defining chunks as feature structures / AVMs induces a natural notion of *identity*, and *information-based ordering*, over the space of all chunks:

- a chunk is identical to another chunk if and only if (iff) they have the same slots and the same values for those slots
- a chunk is a part of (less informative than) another chunk if the latter includes all the ⟨slot, value⟩ pairs of the former and possibly more

The `pyactr` library overloads standard comparison operators for these tasks, as shown below:

[7]:
```
carLexeme2 == carLexeme3
```

[7]:
```
True
```

[8]:
```
carLexeme == carLexeme2
```

[8]:
```
False
```

[9]:
```
carLexeme <= carLexeme2
```

[9]:
```
True
```

[10]:
```
carLexeme < carLexeme2
```

[10]:
```
True
```

[11]:
```
carLexeme2 < carLexeme
```

[11]:
```
False
```

Note that **chunk types are irrelevant for deciding identity or part-of relations**. This might be counter-intuitive, but it's an essential feature of ACT-R:

- chunk types are 'syntactic sugar', useful only for the human modeler

This means that if we define a new chunk type that happens to have the same slots as another chunk type, chunks of one type might be identical to or part of chunks of the other type:

```
[12]: actr.chunktype("syncat", "category")
      anynoun = actr.makechunk(nameofchunk="anynoun1",
                               typename="syncat",
                               category="noun")
      anynoun < carLexeme
```

```
[12]: True
```

```
[13]: anynoun < carLexeme2
```

```
[13]: True
```

This way of defining chunk identity is a direct expression of ACT-R's hypothesis that the **human declarative memory is content-addressable memory**:

- the only way we have to retrieve a chunk is by means of its slot-value content
- chunks are not indexed in any way and cannot be accessed via their index or their memory address
- the only way to access a chunk is by specifying a cue, which is a slot-value pair or a set of such pairs, and retrieving chunks that conform to that pattern, i.e., that are *subsumed* by it

Intuitively, a feature structure, a.k.a. chunk, C_1 **subsumes** another chunk C_2 iff all the information that is contained in C_1 is also contained in C_2.

- we write this as $C_1 \leq C_2$ or $C_1 \sqsubseteq C_2$
- in `pyactr`, we write `C1 <= C2`

Formally, C_1 **subsumes** C_2 iff - all the slots in the domain of C_1 are also in the domain of C_2, and - for each of the slots in the domain of C_1, the value of that slot is *identical* to the value of the corresponding slot in C_2

Not that subsumption in ACT-R (also, in `pyactr`) is not recursively defined, which would require 'is *identical* to' in the previous bullet to be replaced by 'subsumes'.

Discussions and summaries of language-related evidence for content-addressable memory retrieval:

- **McElree, Brian. 2006. "Accessing Recent Events." In *Psychology of Learning and Motivation*, edited by B. H. Ross, 46:155–200. Academic Press.** https://doi.org/https://doi.org/10.1016/S0079-7421(06)46005-9
- **Jäger, Lena A, Felix Engelmann, and Shravan Vasishth. 2017. "Similarity-Based Interference in Sentence Comprehension: Literature Review and Bayesian Meta-Analysis." *Journal of Memory and Language* 94: 316–39**