

Unit 4: Activation of Chunks and Base-Level Learning

There are two goals of this unit. The first is to introduce the subsymbolic quantity of activation associated with chunks. The other is to show how those activation values can be learned through use of the chunks.

4.1 Introduction

We have seen retrieval requests in productions like this:

```
(P example-counting
  =goal>
    isa      count
    state    counting
    number   =num1
  =retrieval>
    isa      count-order
    first    =num1
    second   =num2
==>
  =goal>
    number   =num2
  +retrieval>
    isa      count-order
    first    =num2
)
```

In this case an attempt is being made to retrieve a count-order chunk with a particular number (bound to **=num2**) in its **first** slot. Up to now we have been working with the system at the symbolic level. If there was a chunk that matched that retrieval request it would be placed into the **retrieval** buffer, and if not then the retrieval request would fail and the failure chunk would be placed into the buffer. The system was deterministic and we did not consider any timing cost associated with that retrieval or the possibility a chunk in declarative memory might fail to be retrieved. For the simple tasks we looked at so far that was sufficient.

Most psychological tasks however are not that simple and issues such as accuracy and latency over time or across different conditions are measured. For modeling these more involved tasks one will typically need to use the subsymbolic components of ACT-R to accurately model and predict human performance. For the remainder of the tutorial we will be looking at the subsymbolic components that control the performance of the system. To use the subsymbolic components we need to turn them on by setting the **esc** parameter (enable subsymbolic computations) to **t**:

```
(sgp :esc t)
```

That setting will be necessary for the rest of the models in the tutorial.

4.2 Activation

Every chunk in ACT-R has associated with it a numerical value called its activation. The activation reflects the degree to which past experiences and current context indicate that chunk will be useful at any particular moment. When a retrieval request is made the chunk with the greatest activation among those that match the specification of the request will be the one placed into the retrieval buffer. There is one constraint on that however. There is a parameter called the retrieval threshold which sets the minimum activation a chunk can have and still be retrieved. It is set with the `:rt` parameter in `sgp`:

```
(sgp :rt -0.5)
```

If the chunk with the highest activation among those that match the request has an activation less than the retrieval threshold, then the failure chunk will be placed into the retrieval buffer.

The activation A_i of a chunk i is computed from three components – the base-level, a context component and a noise component. We will discuss the context component in the next unit. So, for now the activation equation is:

$$A_i = B_i + \epsilon$$

B_i : The base-level activation. This reflects the recency and frequency of practice of the chunk i .

ϵ : The noise value. The noise is composed of two components: a permanent noise associated with each chunk and an instantaneous noise computed at the time of a retrieval.

We will discuss these components in detail below.

4.3 Base-level Learning

The equation describing learning of base-level activation for a chunk i is

$$B_i = \ln\left(\sum_{j=1}^n t_j^{-d}\right)$$

n : The number of presentations for chunk i .

t_j : The time since the j th presentation.

d : The decay parameter which is set using the :bll (base-level learning) parameter. This parameter is almost always set to 0.5.

This equation describes a process in which each time an item is presented there is an increase in its base-level activation, which decays away as a power function of the time since that presentation. These decay effects are summed and then passed through a logarithmic transformation.

There are three types of events that are considered a presentation of a chunk. The first is its creation. The second is any merging of that chunk with another and the last is a harvested retrieval of the chunk. The next three subsections describe those events in more detail.

4.3.1 Chunk Creation

When a chunk is initially created is counted as its first presentation. There are a few ways for a chunk to be created, all of which have been discussed in the previous units. They are:

- Explicitly in the initial conditions for the model using the **add-dm** command. These chunks are assumed to be created at time 0, when the model starts.
- Implicitly as the result of a perceptual request. All of the chunks placed into the perceptual buffers (visual-location, visual, aural-location, and aural) are created the first time they are placed into the buffer.
- Explicitly as a new goal using the +goal request on the RHS of a production. This is an important mechanism because it allows a model to create new declarative knowledge while it is running which can be retrieved at some later time. This will play an important role in both of the models for this unit.

4.3.2 Chunk Merging

When a chunk is cleared from the **goal** buffer, either explicitly with a **-goal** action or implicitly with a **+goal** action, it goes through a process we call merging. It is compared to all the other chunks in declarative memory of the same chunk-type. If all of its slot values are an exact match with one of the existing chunks then it is merged with that existing chunk. The chunk being merged is removed from declarative memory and all references to it are changed to references to the existing chunk and the existing chunk is credited with a presentation. This mechanism results in repeated completions of the same operations (the clearing of the duplicate goal chunk) reinforcing the chunk that represents that situation instead of creating lots of identical chunks each with only one presentation.

4.3.3 Retrieval and Harvesting

Retrieving a chunk and then harvesting that retrieval counts as a presentation. Just retrieving the chunk (having it placed into the retrieval buffer) is not enough to count as a presentation. A production that fires must reference the **retrieval** buffer (an =retrieval condition) while that chunk is in the buffer to count as a presentation. In fact every production that fires which references the chunk in the **retrieval** buffer counts as a presentation. Thus, a single retrieval request could result in multiple presentations if multiple productions reference that chunk once it has been put into the **retrieval** buffer. The presentation is counted at the time of the production selection.

4.4 Optimized Learning

Because of the need to separately calculate the effect of each presentation, the learning rule is computationally expensive and for some models that real time cost is too great. To reduce the computational cost there is an approximation that one can use when the presentations are approximately uniformly distributed over the time since the item was created. This approximation can be enabled by turning on the optimized learning parameter - :ol. In fact, its default setting is on (the value **t**). When optimized learning is enabled, the following equation applies:

$$B_i = \ln(n/(1 - d)) - d * \ln(L)$$

n: The number of presentations of chunk *i*.

L: The lifetime of chunk *i* (the time since its creation).

d: The decay parameter.

4.5 Noise

The noise component of the activation equation contains two sources of noise. There is a permanent noise which can be associated with a chunk and an instantaneous noise value which will be recomputed at each retrieval attempt. Both noise values are generated

according to a logistic distribution characterized by a parameter s . The mean of the logistic distribution is 0 and the variance, σ^2 , is related to the s value by this equation:

$$\sigma^2 = \frac{\pi^2}{3} s^2$$

The permanent noise s value is set with the `:pas` parameter and the instantaneous noise s value set with the `:ans` parameter. Typically, we are only concerned with the instantaneous noise (the variance from trial to trial) and leave the permanent noise set off.

4.6 Probability of Recall

If we make a retrieval request and there is a matching chunk, that chunk will only be retrieved if it exceeds the retrieval activation threshold, τ . The probability of this happening depends on the expected activation, A , and the amount of noise in the system which is controlled by the parameter s :

$$\text{recall probability}_i = \frac{1}{1 + e^{\frac{\tau - A_i}{s}}}$$

Inspection of that formula shows that, as A_i tends higher, the probability of recall approaches 1, whereas, as τ tends higher, the probability decreases. In fact, when $\tau = A_i$, the probability of recall is .5. The s parameter controls the sensitivity of recall to changes in activation. If s is close to 0, the transition from near 0% recall to near 100% will be abrupt, whereas when s is larger, the transition will be a slow sigmoidal curve.

4.7 Retrieval Latency

The activation of a chunk also determines how quickly it can be retrieved. When a retrieval request is made the time it takes until the chunk that is retrieved is available in the **retrieval** buffer is given by this equation:

$$\text{Time} = Fe^{-A}$$

A : The activation of the chunk which is retrieved.

F : The latency factor parameter. This parameter is set using the `:lf` parameter.

If no chunk matches the retrieval request, or no chunk has an activation which is greater than the retrieval threshold then a retrieval failure will occur. The time it takes for the failure to be registered in the **retrieval** buffer is:

$$Time = Fe^{-\tau}$$

τ : The retrieval threshold.

F : The latency factor.

4.8 The Paired-Associate Example

Now that we have described how activation works, we will look at an example model which shows the effect of base-level learning. Anderson (1981) reported an experiment in which subjects studied and recalled a list of 20 paired associates for 8 trials. The paired associates consisted of 20 nouns like “house” associated with the digits 0 - 9. Each digit was used as a response twice. Below is the mean percent correct and mean latency to type the digit for each of the trials. Note subjects got 0% correct on the first trial because they were just studying them for the first time and the mean latency is 0 only because there were no correct responses.

Trial	Accuracy	Latency
1	.000	0.000
2	.526	2.156
3	.667	1.967
4	.798	1.762
5	.887	1.680
6	.924	1.552
7	.958	1.467
8	.954	1.402

Anderson, J.R. (1981). Interference: The relationship between response latency and response accuracy. *Journal of Experimental Psychology: Human Learning and Memory*, 7, 326-343.

The Unit 4 folder contains the **paired** model for this experiment. The experiment code is written to allow one to run a general form of the experiment. Both the number of pairs to present and the number of trials to run can be specified. You can run through n trials of m (m no greater than 20) paired associates by calling do-experiment with those parameters:

(do-experiment m n)

For each of the m words you will see the stimulus for 5 seconds during which you have

the opportunity to make your response. Then you will see the associated number for 5 seconds. The simplest form of the experiment is one in which a single pair is presented twice. Here is the trace of the model doing such a task. The first time the model has an opportunity to learn the pair and the second time it has a chance to recall that learned pair:

```
> (do-experiment 1 2)
Time 0.000: Vision found LOC3
Time 0.000: Attend-Probe Selected
Time 0.050: Attend-Probe Fired
Time 0.050: Module :VISION running command MOVE-ATTENTION
Time 0.135: Module :VISION running command ENCODING-COMplete
Time 0.135: Vision sees TEXT0
Time 0.135: Read-Probe Selected
Time 0.185: Read-Probe Fired
Time 0.185: Module :VISION running command CLEAR
Time 0.235: Module :VISION running command CHANGE-STATE
Time 2.771: Failure Retrieved
Time 2.771: Cannot-Recall Selected
Time 2.821: Cannot-Recall Fired
Time 5.000: * Running stopped because time limit reached.
Time 5.000: Vision found LOC6
Time 5.000: Detect-Study-Item Selected
Time 5.050: Detect-Study-Item Fired
Time 5.050: Module :VISION running command MOVE-ATTENTION
Time 5.135: Module :VISION running command ENCODING-COMplete
Time 5.135: Vision sees TEXT5
Time 5.135: Associate Selected
Time 5.185: Associate Fired
Time 10.000: * Running stopped because time limit reached.
Time 10.000: Vision found LOC3
Time 10.085: Module :VISION running command ENCODING-COMplete
Time 10.085: Vision sees TEXT9
Time 10.085: Attend-Probe Selected
Time 10.135: Attend-Probe Fired
Time 10.135: Module :VISION running command MOVE-ATTENTION
Time 10.220: Module :VISION running command ENCODING-COMplete
Time 10.220: Vision sees TEXT9
Time 10.220: Read-Probe Selected
Time 10.270: Read-Probe Fired
Time 10.270: Module :VISION running command CLEAR
Time 10.320: Module :VISION running command CHANGE-STATE
Time 10.395: Goal Retrieved
Time 10.395: Recall Selected
Time 10.445: Recall Fired
Time 10.445: Module :MOTOR running command PRESS-KEY
Time 10.695: Module :MOTOR running command PREPARATION-COMplete
Time 10.745: Module :MOTOR running command INITIATION-COMplete
Time 10.845: Device running command OUTPUT-KEY
Time 10.995: Module :MOTOR running command FINISH-MOVEMENT
Time 15.000: * Running stopped because time limit reached.
Time 15.000: Vision found LOC6
Time 15.000: Detect-Study-Item Selected
Time 15.050: Detect-Study-Item Fired
Time 15.050: Module :VISION running command MOVE-ATTENTION
Time 15.135: Module :VISION running command ENCODING-COMplete
Time 15.135: Vision sees TEXT12
Time 15.135: Associate Selected
Merging chunk Goal8 into chunk Goal
Time 15.185: Associate Fired
Time 20.000: * Running stopped because time limit reached.
```

The basic structure of the screen processing productions should be familiar by now. The one thing to note is that because this model must wait for stimuli to appear on screen it takes advantage of the buffer stuffing mechanism so that it can wait for the change instead of continuously checking. The way it does so is that the first production that will fire for either the probe or the associated number has a **visual-location** buffer test on its LHS and then on the RHS it clears that buffer. Here are the attend-probe and detect-study-item productions for reference:

```
(p attend-probe
  =goal>
    isa      goal
    state    start
    arg1     nil
  =visual-location>
    isa      visual-location
  =visual-state>
    isa      module-state
    modality free
  ==>
  +visual>
    isa      visual-object
    screen-pos =visual-location
  =goal>
    state    attending
  -visual-location>
)
```

```
(p detect-study-item
  =goal>
    isa      goal
    - state  start
    arg1     =a1
  =visual-location>
    isa      visual-location
  =visual-state>
    isa      module-state
    modality free
  ==>
  +visual>
    isa      visual-object
    screen-pos =visual-location
  =goal>
    state    attending
  -visual-location>
)
```

Because the buffer is cleared and no later productions issue a request for a visual-location they must wait for buffer stuffing to put a chunk into the **visual-location** buffer before they can match. Since none of the other productions match in the mean time the model can essentially just wait for the screen to change before doing anything else.

Now we will focus on those productions that are responsible for forming the association and retrieving the chunk. The production that is initially responsible for forming the association is:

```
(p associate
  =goal>
    isa      goal
    state    attending
    arg1     =arg1
  =visual>
    isa      text
    value    =val
==>
  =goal>
    arg2     =val
    state    associated
  +goal>
    isa      goal
    state    start
)
```

This production applies when the goal has already encoded the stimulus in the arg1 slot:

```
Goal
  isa GOAL
  arg1 "zinc"
  arg2 nil
  state attending
```

It stores the response digit in the arg2 slot and results in a chunk like

```
Goal
  isa GOAL
  arg1 "zinc"
  arg2 "9"
  state associated
```

This goal chunk serves as the memory of this trial. The production also sets up a new goal with the **+goal** action to perform the next trial.

The production **read-probe** makes the request for retrieval of such a chunk:

```
(p read-probe
  =goal>
    isa      goal
    state    attending
    arg1     nil
  =visual>
    isa      text
    value    =val
  =visual-state>
    isa      module-state
    modality free
==>
  +retrieval>
    isa      goal
    state    associated
    arg1     =val
```

```

=goal>
  arg1      =val
  state     testing
-visual>
)

```

Then, depending on whether the chunk can be retrieved, one of two production rules may apply:

```

(p recall
  =goal>
    isa      goal
    arg1     =val
    state    testing
  =retrieval>
    isa      goal
    arg1     =val
    arg2     =ans
  =manual-state>
    ISA      module-state
    modality free
==>
  +manual>
    isa      press-key
    key      =ans
  =goal>
    state    read-study-item
)

```

```

(p cannot-recall
  =goal>
    isa      goal
    state    testing
    arg1     =val
  =retrieval>
    isa      error
==>
  =goal>
    state    read-study-item
)

```

The probability of the recall production firing and the mean latency for the recall will be determined by the activation of the chunk that is retrieved and will increase with repeated presentations and harvested retrievals.

The model gives a pretty good fit to the data as illustrated below in a run of 100 simulated subjects (because of stochasticity results are more reliable if there are more runs):

```
> (collect-data 100)
```

Latency:

CORRELATION: 0.999

MEAN DEVIATION: 0.131

Trial	1	2	3	4	5	6	7	8
	0.000	2.019	1.783	1.641	1.514	1.409	1.353	1.314

Accuracy:

CORRELATION: 0.993

MEAN DEVIATION: 0.044

Trial	1	2	3	4	5	6	7	8
	0.000	0.529	0.769	0.856	0.913	0.944	0.940	0.948

4.9 Parameter estimation

To get the model to fit the data requires not only writing a plausible set of productions which can accomplish the task, but also setting the ACT-R parameters that control the behavior as described in the equations governing the operation of declarative memory. Running the model with the default values for the parameters produces the following results:

Latency:

CORRELATION: 0.924

MEAN DEVIATION: 0.247

Trial	1	2	3	4	5	6	7	8
	0.000	1.641	1.625	1.644	1.628	1.632	1.637	1.626

Accuracy:

CORRELATION: 0.884

MEAN DEVIATION: 0.223

Trial	1	2	3	4	5	6	7	8
	0.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000

It shows some of the general trends, but does not fit the data well. The behavior of this model and the one that you have to write really depends on the settings of four parameters. Here are those parameters and their settings in this model. The retrieval threshold is set at -2. This determines how active a chunk has to be to be retrieved 50% of the time. The instantaneous activation noise is set at 0.5. This determines how quickly probability of retrieval changes as we move past the threshold. The latency factor is set at .35. This determines the magnitude of the activation effects on latency. Finally, the decay rate for base-level learning is set to the value .5 which is the where we recommend it be set for most tasks that involve the base-level learning mechanism.

How to determine those values can be a tricky process because the equations are all related and thus they are cannot be independently manipulated for a best fit. Typically some sort of searching is required, and there are many ways to accomplish that. For the tutorial models there will typically be only one or two parameters that you will need to adjust and we recommend that you work through the process “by hand” adjusting the parameters individually to see the effect that they have on the model. However, there are other ways of determining parameters that can be used, but we will not be covering them in the tutorial.

4.10 Unit Exercise: Alpha-Arithmetic

The following data were obtained by N. J. Zbrodoff on judging alphabetic arithmetic problems. Participants were presented with an equation like $A + 2 = C$ and had to respond yes or no whether the equation was correct based on counting in the alphabet –

the preceding equation is correct, but $B + 3 = F$ is not.

She manipulated whether the addend was 2, 3, or 4 and whether the problem was true or false. She had 2 versions of each of the 6 kinds of problems (3 addends x 2 responses) each with a different letter (a through f). She then manipulated the frequency with which problems were studied in sets of 24 trials:

- In the Control condition, each of the 2, 3 and 4 addend problems occurred twice.
- In the Standard condition, the 2 addend problems occurred three times, the 3 addend problems twice, and the 4 addend problems once.
- In the Reverse condition, the 2 addend problems occurred once, the 3 addend problems twice, and the 4 addend problems three times.

Each participant saw problems based on one of the three conditions. There were 8 repetitions of a set of 24 problems in a block (192 problems), and there were 3 blocks for 576 problems in all. The data presented below are in seconds to judge the problems true or false based on the block and the addend. They are aggregated over both true and false responses:

Control Group (all problems equally frequently)

	Two	Three	Four
Block 1	1.840	2.460	2.820
Block 2	1.210	1.450	1.420
Block 3	1.140	1.210	1.170

Standard Group (smaller problems more frequent)

	Two	Three	Four
Block 1	1.840	2.650	3.550
Block 2	1.080	1.450	1.920
Block 3	0.910	1.080	1.430

Reverse Group (larger problems more frequent)

	Two	Three	Four
Block 1	2.250	2.530	2.420
Block 2	1.470	1.460	1.110
Block 3	1.240	1.120	0.870

The interesting phenomenon concerns the interaction between the effect of the addend and amount of practice. Presumably, the addend effect originally occurs because subjects have to engage in counting, but latter they come to rely mostly on retrieval of answers they stored from previous computations.

The task for this unit is to develop a model of the control group data. Functions to run the experiment and most of a model that can perform the task are provided in the model called **zbrodoff**. The model as given does the task by counting through the alphabet and numbers “in its head” (using the subvocalize action of the speech module to produce reasonable timing data) to arrive at an answer which it compares to the initial equation to determine how to respond. Here is the performance of this model on the task:

2 (64) 3 (64) 4 (64)

```
Block 1  2.297 (64)  2.793 (64)  3.283 (64)
Block 2  2.287 (64)  2.798 (64)  3.286 (64)
Block 3  2.297 (64)  2.792 (64)  3.277 (64)
```

It is always correct (64 out of 64 for each cell) but does not get any faster from block to block because it always uses the counting strategy. Your first task is to extend the model so that it attempts to remember previous instances of the trials. If it can remember the answer it does not have to resort to the counting strategy and can respond much faster. The previous trials are encoded in the goal chunks, and encode the result of the counting. A completed goal for a trial where the problem was “B + 3” would look like this:

```
Goal
  isa PROBLEM
  arg1 "b"
  arg2 "3"
  ans nil
  count "3"
  result "e"
  state Count
```

The result slot contains the correct value “E” for “B + 3” as a result of counting. There will be one of these chunks for each of the problems that is encountered, which will be a total of six after it completes the first set of problems.

After your model is able to utilize a retrieval strategy along with the counting strategy given your next step is to adjust the parameters so that the model’s performance better fits the experimental data. The results should look something like this after you have the retrievals working:

```
> (collect-data 1)
CORRELATION:  0.992
MEAN DEVIATION:  0.552

          2 (64)      3 (64)      4 (64)
Block 1  1.326 (64)  1.503 (64)  1.682 (64)
Block 2  1.063 (64)  1.165 (64)  1.081 (64)
Block 3  1.073 (64)  1.051 (64)  1.042 (64)
```

The model’s correlation is good, but the deviation is quite high because the model is too fast overall. The model’s performance will depend on the same four parameters as the paired associate model: latency factor, activation noise, base-level decay rate, and retrieval threshold. In the model you are given, the first three are set to the same values as in the paired associate model and represent reasonable values for this task. You should not have to adjust any of those. However, the retrieval threshold (the `:rt` parameter) is set to its default value of 0. This is the parameter you should manipulate to improve the fit to the data. Here is our fit to the data adjusting only the retrieval threshold:

```
> (collect-data 40)
CORRELATION:  0.990
MEAN DEVIATION:  0.171
          Two (64)    Three (64)    Four (64)
Block 1:  1.856 (64)  2.121 (64)  2.480 (64)
Block 2:  1.317 (64)  1.424 (64)  1.516 (64)
Block 3:  1.172 (64)  1.228 (64)  1.271 (64)
```

This experiment is more complicated than the ones that you have seen previously. It runs continuously for many trials and the learning that occurs across trials is important. Thus the model cannot treat each trial as an independent event and be reset before each one as has been done for the previous units. While writing your model and testing the fit to the data you will probably want to test it on smaller runs than the whole task. There are four functions you can use to test the experiment. The first is **collect-data** which takes one parameter, the number of times to run the full experiment. That function will average the results of running the full experiment multiple times and report the correlation and deviation to the experimental data. It may take a long time to run, especially if you request a lot of runs for comparing to the data (to speed up the runs once you are sure the model is doing the right thing you can turn off the trace, the `:v` parameter in the `sgp` command). You can use the **do-block** function, which takes one optional parameter, to run 1 block of the experiment and display the results. The optional parameter controls whether or not to show the window while the task is running. If you do not supply the parameter a virtual window is used, and if you specify the parameter `t` then a real window will be shown. The **do-set** function is similar to **do-block**, except it only runs through the set of 24 items once. The **do-trial** function can be used to run a single trial. It takes four parameters which are all single character strings and an optional fifth parameter like **do-block** and **do-set** to control the display. The first three are the elements of the equation to present i.e. "a" "2" "c" to present $a + 2 = c$. The fourth is the correct key press, "K" for a true probe and "D" for a false probe. One thing to note is that the only one of those functions to call **reset** is **collect-data**. So if you are using the other functions while testing the model keep in mind that unless you call **reset** it will remember everything it has done since the last time it was reset.

There are two commands in the commands window of the provided model that you have not seen before:

```
(setallbaselevels 10000000 -1000)
(add-dm-fct '((goal isa problem)))
```

The first one sets the base-level activation of all the chunks that exist when it is called (which are the sequence chunks provided) to very large values by setting the parameters **n** and **L** of the optimized base-level equation for each one. The first parameter, 10000000, specifies **n** and the second parameter, -1000, specifies the creation time of the chunk. This ensures that they maintain a very high base-level activation and do not fall below the retrieval threshold over the course of the task.

The second call creates the initial goal chunk for the model. We do not want it to have a high base-level like the sequence chunks, so it must be created after the call to `setallbaselevels`. Because the ACT-R environment editor separates the components of the model into separate windows (by default) if we were to just use `add-dm` to create it the environment could put it in the chunks window and it would be created before the `setallbaselevels` command gets called. So instead, we use the functional form of the `add-dm` command to ensure it stays where we put it when using the environment (that is not

the only difference between add-dm and add-dm-fct, but is the only one of consequence for the tutorial).

Zbrodoff, N. J. (1995). Why is $9 + 7$ harder than $2 + 3$? Strength and interference as explanations of the problem-size effect. *Memory & Cognition*, 23 (6), 689-700.