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 are learned through the history of usage 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 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 buffer would indicate the failure. The system was deterministic and we did not consider any timing cost associated with that retrieval or the possibility that a matching chunk in declarative memory might fail to be retrieved. For the simple tasks we have 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's declarative memory 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:

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

If the chunk with the highest activation among those that match the request has an activation which is less than the retrieval threshold, then no chunk will be placed into the **retrieval** buffer and a buffer failure will be indicated.

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:

$$A_i = B_i + \varepsilon$$

$\boldsymbol{B_i}$**:** The base-level activation. This reflects the recency and frequency of practice of the chunk *i*.

$\boldsymbol{\varepsilon}$: 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 request.

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*.
$\boldsymbol{t_j}$: The time since the *jth* 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 two types of events that are considered as presentations of a chunk. The first is its initial entry into declarative memory. The other is when a chunk merges with a chunk that is already in declarative memory. The next two subsections describe those events in more detail.

4.3.1 Chunks Entering Declarative Memory

When a chunk is initially entered into declarative memory is counted as its first presentation. There are two ways for a chunk to be entered into declarative memory, both of which have been discussed in the previous units. They are:

- Explicitly by the modeler using the **add-dm** command. These chunks are entered at the time the call is executed, which is time 0 for a call in the body of the model definition.

- When the chunk is cleared from a buffer. We have seen many of the previous models clearing buffers, and when that happens, the cleared chunk can then be found among the chunks in declarative memory.

4.3.2 Chunk Merging

Something we have not seen previously is what happens when the chunk cleared from a buffer is an identical match to a chunk which is already in declarative memory. If a chunk has the same set of slots and values as a chunk which already exists in the model's declarative memory then instead of being added to declarative memory that chunk goes through a process we refer to as merging with the existing chunk in declarative memory. Instead of adding the new chunk to declarative memory the preexisting chunk in declarative memory is credited with a presentation. Then the name of the new chunk (the one which is being merged into declarative memory) is changed to now reference the chunk that was already in declarative memory i.e. there is one chunk which now has two (or possibly more) names. This mechanism results in repeated completions of the same operations (the clearing of a duplicate chunk) reinforcing the chunk that represents that situation instead of creating lots of identical chunks each with only one presentation. So, for example, repeatedly attending the same visual stimuli would result in strengthening a single chunk that represents that object.

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 the real time cost of computation is too great to be able to actually run the model in a reasonable amount time. 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:

$$B_i = \ln\left(\frac{n}{1-d}\right) - d * \ln(L)$$

***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 is 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 turned off (a value of **nil**).

4.6 Probability of Recall

If we make a retrieval request and there is a matching chunk, that chunk will only be retrieved if its activation exceeds the retrieval threshold, τ. The probability of this happening depends on the expected activation of the chunk (its activation without the noise added in), A_i, and the amount of noise in the system which is controlled by the parameter *s*:

$$recallprobability_i = \frac{1}{1+e^{\frac{\tau - A_i}{s}}}$$

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. It is important to note however that this is only a description of what can happen with the retrieval of a chunk. When a retrieval request is made the chunk's activation plus the instantaneous noise will either be above the threshold or not.

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:

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

A: The activation of the chunk which is retrieved.

F: The latency factor (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 signaled is:

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

$\boldsymbol{\tau}$: 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

The paired.lisp file contains a version of this experiment and a model. 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 the model through n trials of m paired associates (m no greater than 20) by calling paired-task with those parameters:

```
(paired-task m n)
```

If you would like to do the task as a person, include the symbol human as the third parameter to paired-task:

```
(paired-task m n 'human)
```

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:

```
> (paired-task 1 2)
 0.000   GOAL         SET-BUFFER-CHUNK GOAL GOAL REQUESTED NIL
 0.000   VISION       SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0-0 REQUESTED NIL
 0.000   PROCEDURAL   CONFLICT-RESOLUTION
 0.050   PROCEDURAL   PRODUCTION-FIRED ATTEND-PROBE
 0.050   PROCEDURAL   CLEAR-BUFFER VISUAL-LOCATION
 0.050   PROCEDURAL   CLEAR-BUFFER VISUAL
 0.050   PROCEDURAL   CONFLICT-RESOLUTION
 0.135   VISION       Encoding-complete VISUAL-LOCATION0-0-0 NIL
 0.135   VISION       SET-BUFFER-CHUNK VISUAL TEXT0
 0.135   PROCEDURAL   CONFLICT-RESOLUTION
 0.185   PROCEDURAL   PRODUCTION-FIRED READ-PROBE
 0.185   PROCEDURAL   CLEAR-BUFFER VISUAL
 0.185   PROCEDURAL   CLEAR-BUFFER IMAGINAL
 0.185   PROCEDURAL   CLEAR-BUFFER RETRIEVAL
 0.185   DECLARATIVE  START-RETRIEVAL
 0.185   PROCEDURAL   CONFLICT-RESOLUTION
 0.385   IMAGINAL     SET-BUFFER-CHUNK IMAGINAL CHUNK0
 0.385   PROCEDURAL   CONFLICT-RESOLUTION
 3.141   DECLARATIVE  RETRIEVAL-FAILURE
 3.141   PROCEDURAL   CONFLICT-RESOLUTION
 3.191   PROCEDURAL   PRODUCTION-FIRED CANNOT-RECALL
 3.191   PROCEDURAL   CLEAR-BUFFER VISUAL
```

```
 3.191   VISION       CLEAR
 3.191   PROCEDURAL   CONFLICT-RESOLUTION
 3.241   PROCEDURAL   CONFLICT-RESOLUTION
 5.000   ------       Stopped because time limit reached
 5.000   VISION       SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION1-0 REQUESTED NIL
 5.000   PROCEDURAL   CONFLICT-RESOLUTION
 5.050   PROCEDURAL   PRODUCTION-FIRED DETECT-STUDY-ITEM
 5.050   PROCEDURAL   CLEAR-BUFFER VISUAL-LOCATION
 5.050   PROCEDURAL   CLEAR-BUFFER VISUAL
 5.050   PROCEDURAL   CONFLICT-RESOLUTION
 5.135   VISION       Encoding-complete VISUAL-LOCATION1-0-0 NIL
 5.135   VISION       SET-BUFFER-CHUNK VISUAL TEXT1
 5.135   PROCEDURAL   CONFLICT-RESOLUTION
 5.185   PROCEDURAL   PRODUCTION-FIRED ASSOCIATE
 5.185   PROCEDURAL   CLEAR-BUFFER IMAGINAL
 5.185   PROCEDURAL   CLEAR-BUFFER VISUAL
 5.185   VISION       CLEAR
 5.185   PROCEDURAL   CONFLICT-RESOLUTION
 5.235   PROCEDURAL   CONFLICT-RESOLUTION
10.000   ------       Stopped because time limit reached
10.000   VISION       SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION2-0 REQUESTED NIL
10.000   PROCEDURAL   CONFLICT-RESOLUTION
10.050   PROCEDURAL   PRODUCTION-FIRED ATTEND-PROBE
10.050   PROCEDURAL   CLEAR-BUFFER VISUAL-LOCATION
10.050   PROCEDURAL   CLEAR-BUFFER VISUAL
10.050   PROCEDURAL   CONFLICT-RESOLUTION
10.135   VISION       Encoding-complete VISUAL-LOCATION2-0-0 NIL
10.135   VISION       SET-BUFFER-CHUNK VISUAL TEXT2
10.135   PROCEDURAL   CONFLICT-RESOLUTION
10.185   PROCEDURAL   PRODUCTION-FIRED READ-PROBE
10.185   PROCEDURAL   CLEAR-BUFFER VISUAL
10.185   PROCEDURAL   CLEAR-BUFFER IMAGINAL
10.185   PROCEDURAL   CLEAR-BUFFER RETRIEVAL
10.185   DECLARATIVE  START-RETRIEVAL
10.185   PROCEDURAL   CONFLICT-RESOLUTION
10.385   IMAGINAL     SET-BUFFER-CHUNK IMAGINAL CHUNK1
10.385   PROCEDURAL   CONFLICT-RESOLUTION
11.145   DECLARATIVE  RETRIEVED-CHUNK CHUNK0-0
11.145   DECLARATIVE  SET-BUFFER-CHUNK RETRIEVAL CHUNK0-0
11.145   PROCEDURAL   CONFLICT-RESOLUTION
11.195   PROCEDURAL   PRODUCTION-FIRED RECALL
11.195   PROCEDURAL   CLEAR-BUFFER RETRIEVAL
11.195   PROCEDURAL   CLEAR-BUFFER MANUAL
11.195   PROCEDURAL   CLEAR-BUFFER VISUAL
11.195   MOTOR        PRESS-KEY KEY 9
11.195   VISION       CLEAR
11.195   PROCEDURAL   CONFLICT-RESOLUTION
11.245   PROCEDURAL   CONFLICT-RESOLUTION
11.445   PROCEDURAL   CONFLICT-RESOLUTION
11.495   PROCEDURAL   CONFLICT-RESOLUTION
11.595   MOTOR        OUTPUT-KEY #(9 2)
11.595   PROCEDURAL   CONFLICT-RESOLUTION
11.745   PROCEDURAL   CONFLICT-RESOLUTION
15.000   ------       Stopped because time limit reached
15.000   VISION       SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION3-0 REQUESTED NIL
15.000   PROCEDURAL   CONFLICT-RESOLUTION
15.050   PROCEDURAL   PRODUCTION-FIRED DETECT-STUDY-ITEM
15.050   PROCEDURAL   CLEAR-BUFFER VISUAL-LOCATION
15.050   PROCEDURAL   CLEAR-BUFFER VISUAL
15.050   PROCEDURAL   CONFLICT-RESOLUTION
15.135   VISION       Encoding-complete VISUAL-LOCATION3-0-0 NIL
15.135   VISION       SET-BUFFER-CHUNK VISUAL TEXT3
15.135   PROCEDURAL   CONFLICT-RESOLUTION
```

```
15.185   PROCEDURAL  PRODUCTION-FIRED ASSOCIATE
15.185   PROCEDURAL  CLEAR-BUFFER IMAGINAL
15.185   PROCEDURAL  CLEAR-BUFFER VISUAL
15.185   VISION      CLEAR
15.185   PROCEDURAL  CONFLICT-RESOLUTION
15.235   PROCEDURAL  CONFLICT-RESOLUTION
20.000   ------      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 that is by having the first production that will match, for either the probe or the associated number, have a **visual-location** buffer test on its LHS and no productions which make requests for visual-locations. Thus, those productions will only match once buffer stuffing places a chunk into the **visual-location** buffer. Here are the **attend-probe** and **detect-study-item** productions for reference:

```
(p attend-probe
    =goal>
      isa      goal
      state    start
    =visual-location>
    ?visual>
     state     free
   ==>
    +visual>
      cmd      move-attention
      screen-pos =visual-location
    =goal>
      state    attending-probe
)


(p detect-study-item
    =goal>
      isa      goal
      state    read-study-item
    =visual-location>
    ?visual>
      state    free
   ==>
    +visual>
      cmd      move-attention
      screen-pos =visual-location
    =goal>
      state    attending-target
)
```

Because the buffer is cleared automatically by strict harvesting and no later productions issue a request for a visual-location these productions 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 will essentially just wait for the screen to change before doing anything else.

Now we will focus on the productions which are responsible for forming the association and retrieving the chunk. When the model attends to the probe with the **read-probe** production two actions are taken (in addition to the updating of the goal state):

```
(p read-probe
    =goal>
      isa      goal
      state    attending-probe
    =visual>
      isa      visual-object
      value    =val
    ?imaginal>
      state    free
   ==>
    +imaginal>
      isa      pair
      probe    =val
    +retrieval>
      isa      pair
      probe    =val
    =goal>
      state    testing
)
```

It makes a request to the **imaginal** buffer to create a chunk which will hold the value read from the screen in the probe slot. It also makes a request to the **retrieval** buffer to retrieve a chunk from declarative memory which has that same value in the probe slot.

We will come back to the retrieval request shortly. For now we will focus on the creation of the chunk for representing the pair of items in the **imaginal** buffer.

The **associate** production fires after the model reads the number which is associated with the probe:

```
(p associate
    =goal>
      isa      goal
      state    attending-target
    =visual>
      isa      visual-object
      value    =val
```

```
   =imaginal>
      isa      pair
      probe    =probe
    ?visual>
      state    free
  ==>
   =imaginal>
      answer   =val
   -imaginal>
   =goal>
      state    start
   +visual>
      cmd      clear
)
```

This production sets the answer slot of the chunk in the **imaginal** buffer to the answer which was read from the screen. It also then clears that chunk from the buffer so that it is entered into declarative memory. That will result in a chunk like this being added to the model's declarative memory:

```
CHUNK0-0
   PROBE  "zinc"
   ANSWER  "9"
```

This chunk serves as the memory of this trial. An important thing to note is that the chunk in the buffer is not added to the model's declarative memory until that buffer is cleared. Often that happens when the model later harvests that chunk from the buffer, but in this case the model does not harvest the chunk later so it is explicitly cleared in the last production which modifies it. One could imagine adding additional productions which would rehearse that chunk, but for the demonstration model that is not done.

This production also makes a request to the **visual** buffer to stop attending to the item. That is done so that the model does not perform the automatic re-encoding when the screen is updated.

Now, consider the retrieval request in the read-probe production again:

```
    +retrieval>
      isa      pair
      probe    =val
```

The declarative memory module will attempt to retrieve a chunk with the requested probe value. Depending on whether a chunk can be retrieved, one of two production rules may apply corresponding to either the successful retrieval of such a chunk or the failure to retrieve a matching chunk:

```
(p recall
    =goal>
```

```
      isa      goal
      state    testing
    =retrieval>
      isa      pair
      answer   =ans
    ?manual>
      state    free
    ?visual>
      state    free
   ==>
    +manual>
      cmd      press-key
      key      =ans
    =goal>
      state    read-study-item
    +visual>
      cmd      clear)

(p cannot-recall
    =goal>
      isa      goal
      state    testing
    ?retrieval>
      buffer   failure
    ?visual>
      state    free
   ==>
    =goal>
     state    read-study-item
    +visual>
     cmd      clear)
```

The probability of the recall production firing and the mean latency for the recall will be determined by the activation of the corresponding trial chunk. The probability will increase with repeated presentations and successful retrievals, while the latency will decrease. That is because each repeated presentation will result in creating a new chunk in the **imaginal** buffer which will merge with the existing chunk for that trial in declarative memory thus increasing its activation. Similarly, when it successfully retrieves a chunk for a trial and the recall production fires that chunk in the **retrieval** buffer will be cleared by the strict harvesting mechanism and then merge with the chunk in declarative memory again increasing its activation.

This 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 and to generate that many runs in a reasonable amount of time one must turn off the trace and remove the seed parameter to allow for differences from run to run):

```
> (paired-experiment 100)

Latency:
CORRELATION:  0.998
MEAN DEVIATION:  0.097
Trial    1       2       3       4       5       6       7       8
       0.000   2.139   1.834   1.665   1.546   1.440   1.377   1.306


Accuracy:
CORRELATION:  0.994
MEAN DEVIATION:  0.043
Trial    1       2       3       4       5       6       7       8
       0.000   0.554   0.769   0.855   0.904   0.932   0.955   0.960
```

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:

```
> (paired-experiment 5)


Latency:
CORRELATION:  0.000
MEAN DEVIATION:  1.619
Trial    1       2       3       4       5       6       7       8
       0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000


Accuracy:
CORRELATION:  0.000
MEAN DEVIATION:  0.777
Trial    1       2       3       4       5       6       7       8
       0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
```

which shows a complete failure to retrieve any of the facts. Just lowering the retrieval threshold so that they can be retrieved results in something like this:

```
> (paired-experiment 5)

Latency:
CORRELATION:  0.972
MEAN DEVIATION:  1.187
Trial    1       2       3       4       5       6       7       8
       0.000   3.512   3.871   3.359   2.793   2.473   2.265   2.136


Accuracy:
CORRELATION:  0.914
MEAN DEVIATION:  0.176
Trial    1       2       3       4       5       6       7       8
       0.000   0.100   0.600   1.000   1.000   1.000   1.000   1.000
```

That 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 0.4. This determines the magnitude of the activation effects on latency. Finally, the decay rate for base-level learning is set to the value 0.5 which is 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 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. There are other ways of determining parameters that can be used, but we will not be covering any such mechanisms in the tutorial.

4.10 The Activation trace

A parameter named :act is also set in the sgp call of the **paired** model. This is the activation trace parameter. If it is turned on, it causes the declarative memory system to print the details of the activation computations that occur during a retrieval request in the trace. If you set it to **t** and reload the model and run for two trials of one pair (like the trace above) you will find these additional details where the retrievals occur:

```
0.185   DECLARATIVE START-RETRIEVAL
No matching chunk found retrieval failure
...
10.185   DECLARATIVE START-RETRIEVAL
Chunk CHUNK0-0 matches
Computing activation for chunk CHUNK0-0
Computing base-level
Starting with blc: 0.0
Computing base-level from 1 references (5.185)
  creation time: 5.185 decay: 0.5  Optimized-learning: T
base-level value: -0.11157179
Total base-level: -0.11157179
Adding transient noise -0.7636829
Adding permanent noise 0.0
Chunk CHUNK0-0 has an activation of: -0.8752547
Chunk CHUNK0-0 has the current best activation -0.8752547
Chunk CHUNK0-0 with activation -0.8752547 is the best
...
```

You may find this detailed accounting of the activation computation useful in debugging your models or just in understanding how the system computes activation values.

4.11 The :ncnar Parameter

There is one final parameter being set in the model which you have not seen before - :ncnar (normalize chunk names after run). This parameter does not affect the model's performance on the tasks, but it does affect the actual time it takes to run the simulation. The details on what exactly it does can be found in the experiment text for this unit. The reason it is turned off (set to **nil**) in the models for this unit is to speed up the time it takes to run the simulations.

4.12 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   1.060   1.450   1.920
Block 3   0.910   1.080   1.480

Reverse Group (larger problems more frequent)
           Two    Three    Four
Block 1   2.250   2.530   2.440
Block 2   1.470   1.460   1.100
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 have 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:

```
> (zbrodoff 1)
CORRELATION:  0.289
MEAN DEVIATION:  1.309

                2 (64)      3 (64)      4 (64)
Block  1  2.301 (64)  2.806 (64)  3.287 (64)
Block  2  2.290 (64)  2.804 (64)  3.301 (64)
Block  3  2.286 (64)  2.797 (64)  3.290 (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 model encodes each trail in a chunk which has the result of its counting for the trial. A completed problem for a trial where the stimulus was "A+2 = C" would look like this:

```
CHUNK0-0
   RESULT  "c"
   ARG1  "a"
   ARG2  "2"
```

The result slot contains the result of counting 2 letters from A. An important thing to note is that the actual target letter for the trial is stored in the goal buffer for comparison after the model has finished counting to a result. The model only encodes the result of the counting in the chunk that represents a trial in the imaginal buffer. Thus the same chunk will result from a trial where the stimulus presented is "A+2 = D" because it only counts A plus 2. The assumption is that the person is actually learning the letter counting facts and not just memorizing the stimulus-response pairings for the task. The model will learn one chunk for each of the additions which it encounters, which will be a total of six after it completes a set of trials.

A strong recommendation for adding the retrieval strategy to the model is to continue to use the existing encoding productions before the retrieval, the existing response productions (**final-answer-yes** and **final-answer-no**) after a successful retrieval, and the given counting productions if it fails to retrieve. It may be necessary to modify productions at the end of the encoding process and/or the beginning of the counting process to add the retrieval process into the model, but the response productions should not be modified in any way. Using the given response productions is important because they already handle the important steps necessary for the model to repeatedly perform this task successfully: they create a new goal chunk which will make sure the model is ready for the next trial, they make the correct response based on the comparison of the value read from the screen and the correct value of the sum which has been encoded in the imaginal buffer, and that imaginal buffer chunk is cleared so that it can enter declarative memory and strengthen the knowledge for that fact.

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 retrieval strategy working with the parameters as set in the starting model:

```
CORRELATION:  0.929
MEAN DEVIATION:  0.656

              2 (64)      3 (64)      4 (64)
Block  1  1.265 (64)  1.444 (64)  1.338 (64)
Block  2  1.094 (64)  1.077 (64)  1.093 (64)
Block  3  1.043 (64)  1.047 (64)  1.039 (64)
```

The model is still always responding correctly on all trials, the 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. The retrieval threshold (the :rt parameter) is set to its default value of 0. This is the parameter you should manipulate first to improve the fit to the data. Here is our fit to the data adjusting only the retrieval threshold:

```
> (zbrodoff 20)
CORRELATION:  0.987
MEAN DEVIATION:  0.174

              2 (64)      3 (64)      4 (64)
Block  1  1.870 (64)  2.179 (64)  2.558 (64)
Block  2  1.376 (64)  1.520 (64)  1.632 (64)
Block  3  1.215 (64)  1.282 (64)  1.359 (64)
```

If you would like to try to fit the data even better then you could also adjust the latency factor and activation noise parameters as well. The base-level decay rate parameter should be left at the value .5 (that is a recommended value which should not be adjusted in most models). Here is our best fit with adjusting all three parameters:

```
> (zbrodoff 300)
CORRELATION:  0.991
MEAN DEVIATION:  0.082

              2 (64)      3 (64)      4 (64)
Block  1  1.994 (64)  2.375 (64)  2.755 (64)
Block  2  1.267 (64)  1.361 (64)  1.448 (64)
Block  3  1.092 (64)  1.110 (64)  1.132 (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 run parts of the experiment.

The **zbrodoff-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. 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 which should be pressed for the trial, "K" for a true probe and "D" for a false probe. The optional parameter indicates whether or not to show the task display. If the optional parameter is not provided then the window will not be shown (a virtual window will be used) and if it is the value **t** then it will show the task window. This call would present the a + 2 = c problem to the model with a window that is visible:

```
(zbrodoff-trial "a" "2" "c" "k" t)
```

Here are the twelve different problems which are used in this experiment along with the correct response for each:

```
("a" "2" "c" "k")("d" "2" "f" "k")
("b" "3" "e" "k")("e" "3" "h" "k")
("c" "4" "g" "k")("f" "4" "j" "k")
("a" "2" "d" "d")("d" "2" "g" "d")
("b" "3" "f" "d")("e" "3" "i" "d")
("c" "4" "h" "d")("f" "4" "k" "d")
```

The zbrodoff-trial function should be used until you are certain that your model is able to successfully use a retrieval strategy along with counting. To do that you will want to present the model with the same trial again and again and make sure that at some point it can retrieve the correct fact and respond correctly. You will also want to test it with both true and false facts to make sure it can retrieve the right information and respond correctly in both cases. You will also want to check the model’s declarative memory to make sure that it is only creating the correct facts – it should not be learning chunks which represent the wrong addition.

Once you are confident that your model is learning the correct chunks and can use both retrieval and counting to respond correctly you can use the **zbrodoff-set** and **zbrodoff-block** functions to run the model over multiple trials. Each takes one optional parameter, like **zbrodoff-trial**, to control whether the task display is shown, and if it is not provided they will not show the display. The **zbrodoff-set** function runs the model through 24 trials of the task presenting each problem twice in a randomly generated order. The **zbrodoff-block** function runs through 192 trials, which is 8 repetitions of the 24 trial set. The data which is being modeled is the result of three blocks of trials.

After making sure the model can successfully complete a set and block of trials then you will want to use the **zbrodoff** function to run it through the experiment multiple times and compare its performance to the data. The **zbrodoff** function takes one parameter indicating the number of times to run the full experiment. That function will average the results of running the full experiment that many times and report the correlation and deviation to the experimental data. It may take a while to run, especially if you request a lot of trials.

An important thing to note is that the only one of those functions which calls **reset** is **zbrodoff**. So if you are using the other functions while testing the model keep in mind that unless you call the **reset** function, press the "Reset" button on the Control Panel, or reload the model, then the model will still have all the chunks which it has learned since the last time it was reset (or loaded) in its declarative memory.

As you look at the starting model you will see one additional setting at the end of the model definition which you have not seen before:

```
(set-all-base-levels 100000 -1000)
```

This sets the base-level activation of all the chunks in declarative memory 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, 100000, specifies **n** and the second parameter, -1000, specifies the creation time of the chunk. This ensures that the initial chunks which encode the sequencing of numbers and letters maintain a very high base-level activation and do not fall below the retrieval threshold over the course of the task. The assumption is that counting and the order of the alphabet are very well learned tasks for the model and the human participants and the use of that knowledge does not lead to any significant learning for those things during the course of the experiment.

Finally, because this experiment involves a lot of trials and you need to run several experiments to get the average results of the model there are some additional things that can be done to improve the performance of the software simulation itself i.e. the real time it takes to run the model through the experiment not the simulated time the model reports for doing the task. Probably the most important will be to turn off the model's trace by setting the :v parameter to **nil**. The starting model has that setting, but while you are testing and debugging your addition of a retrieval process you will probably want to turn it back on by setting it to **t** so that you can see what is happening in your model. Something else which you will want to do when running the whole experiment is to close any open inspector tools in the ACT-R Environment because they update with every change to ACT-R and thus will slow down the running of the system. One final thing which can be done is to compile the model file to improve the performance of the Lisp code which performs the task itself. Some Lisp systems do this automatically (Clozure Common Lisp for example does) but many do not. Doing so in a Lisp which does not automatically compile may result in a further reduction in the time it takes to run the experiment. Some details on how to compile a model file and potential problems to be careful of when doing so can be found in the experiment document for this unit.

References

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

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.