# Modeling of Transfer in Complex Tasks

Niclas Andreas Dobbertin
Technische Universität Darmstadt

## Abstract

ABSTRACT

## Modeling of Transfer in Complex Tasks

## Introduction

Transfer learning is the ability to apply lessons learning from one task, to another related or even unrelated task. Living in a complex environment like the real world, a plethora of different tasks like navigating areas, finding things visually or preparing a meal have to be done.

much more efficient if knowledge from tasks can be reused in other tasks

Cognitive Architectures, modeling learning, production systems, ACT-R, frensch task

## Productions

Productions decide how a production system behaves and what actions it takes. A production consists of two parts, a condition and an action (Table 1). All statements listed in the condition must be fulfilled to make the production eligible for selection. In ACT-R, conditions check for specific variable values most of the time, but can also check if certain buffers are empty, full or had an error, e.g. when failing to retrieve something from declarative memory. Only productions which have their conditions satisfied by the current state of the model can be selected by it. Once a production has been selected, its action will be executed. Productions in ACT-R change values of variables and start visual, motor and memory related processes.

If the conditions of multiple productions are satisfied, ACT-R chooses the production with the highest utility. Each production starts with a baseline utility value, which gets updated by the model during its runtime.

**Table 1**
*Example Production*

| IF |
| --- |
| variable1 = true |
| variable2 = 10 |
| **THEN** |
| variable2 = 9 |
| press button |

*Note.* A production consists of two parts: 1. The conditions (**IF**), which must be fulfilled for the production to be available for selection. 2. The actions (**THEN**), which are performed when the production is selected.

## Learning

Retrieval(activation) strength, utility learning, production compilation, . . .

There are a variety of methods production systems use to model learning. ACT-R can adjust which production is given preference during selection or create new productions based on existing ones and the models state.

When multiple productions are applicable to the current state, the production that the model thinks is the most useful should be selected. How useful a production is

can be learned while the model is running and is modeled in ACT-R through a reinforcement learning like process called utility learning.

Oftentimes a series of productions need to be executed in order, this can be combined in to a single production which does all of the actions at once, saving time deciding on which production to use. This method is called production compilation. When two productions are successfully called in a row, a production compilation process is started and combines both into a single production, if possible. Since the compiled productions are specific to the buffer values when the compilation was done, there can be many different combined productions of the same two productions. E.g. a production starting retrieval of an addition fact and a production using the retrieved fact can combine into specific addition-result combinations, skipping retrieval (Shown in Table 2).

(do stuff allegory? learning general production from specific ones (not used))

ACT-Rs subsymbolic system also models delays and accuracy of the declarative memory, where retrieving memories can fail based on their activation strength. Activation strength increases the more often a memory is created or retrieved. Learning new facts and increasing their activation strength is also part of the learning process in an ACT-R model.

(a) *Production 1*

| **IF** |
| operation = subtract |
| argument1 = x |
| argument2 = y |
| **THEN** |
| retrieve: x - y |

(b) *Production 2*

| **IF** |
| operation = subtract |
| retrieve = z |
| **THEN** |
| press button: z |

(c) *A Compiled Production*

| **IF** |
| operation = subtract |
| argument1 = 3 |
| argument2 = 1 |
| **THEN** |
| press button: 2 |

**Table 2**

*Production Compilation*

*Note.* Table 2a shows a production with the condition that the operation variable must be "subtract", and argument1 and 2 must have any values x and y. If selected, it starts retrieval of the result of $x - y$ from declarative memory. Production 2 (Table 2b) is selected when the operation value is subtract as well, and the retrieval variable is filled with a value z. It then starts a motor process to press button z. When the model executes both productions after another, it starts the production compilation process with the current model state. E.g. in Table 2c, if argument1 was 3 and argument2 was 1, it creates a new production which skips retrieval to directly press the result, if the same model state happens again. That means for each combination of x, y and z a different specific production can be created.

**Task**

To investigate model behavior and potentially compare it to results from human experiments, it was decided to use an adapted version of the setup described in Frensch (1991), which was first used in Elio (1986). Subjects are put in charge of determining the quality of water samples by performing simple mathematical operations with given indicator values per water sample. A water sample has an algae, a solids and multiple

toxin and sandstone values, which are randomly generated for each sample. There are six different 2-step equations that use these values and a seventh equation using all previously calculated results to determine the final result (see Table 3). To solve a procedure, subjects have to locate the values of used variables on the screen. Some variables show multiple values, procedures using them indicate how it should be selected after an underscore. For example x_2 means taking the second value of variable x. Other procedures require finding the maximum or minimum value of a variable or of previous solutions. An example of how the screen could look during a trial is shown in Figure 1.

The experiment starts with 75 training trials, each representing a water sample, in which a random choice of 6 procedures has to be solved in the order they are presented. The last procedure is always picked in the selection process, as it uses all previous results for a water sample to calculate the final solution. Afterwards 50 testing trials take place, in which the third procedure from the training phase is switched for the unpicked one. There are three conditions that determine the order in which procedures are presented in the training phase, however the procedure for the final result is always last. In the fixed condition, the order is randomized once at the start and stays constant during all trials. In the random condition, procedure order is randomized between each trial. In the blocked condition, the first procedure has to be solved for all trials before moving on to the second procedure, etc. The testing phase always uses fixed order.

How modeled: Improvements in task performance are mainly dependent on production compilation, as the order and how efficiently the mathematical operations are performed are the main subject of the task. Utility learning matters mostly on production selection and ordering, however the task itself is mostly linear. It can still play a significant role if alternative or shortcut productions for mathematical operations exist. E.g. a production that swaps argument 1 and argument 2 in addition or multiplication may reduce time spend, dependent on how the algorithm functions.

The subsymbolic system of ACT-R also involves mechanisms to gauge retrieval chance and activation strength in the declarative memory. This is used to model learning and retrieval of new memory chunks. In this task however, the subject already has knowledge of mathematical facts and ⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯

"not learn new facts really during exp"

**Figure 1**

*Screenshot of experiment display*

| Mineralien | Algen | | Sandstein | Gifte |
|---|---|---|---|---|
| 5 | 4 | | 9 | 3 |
| | | | 4 | 1 |
| | | | 5 | 2 |
| | | | 7 | 2 |

| | | |
|---|---|---|
| Das Kleinere von (Sandstein_1 + Gifte_1), (Algen) | = | 04 |
| (2 * Algen) + Sandstein_min | = | 12 |
| (Mineralien * 2) - Gifte_4 | = | 08 |
| (Sandstein_4 - Sandstein_2) * Mineralien | = | 15 |
| Das Höhere von (Gifte_3 - Gifte_2), (Sandstein_3) | = | 05 |
| 100 - dem Höchstem aller Ergebnisse | = | |

*Note.* Example water sample presenting in an experiment using the adapted task from Frensch (1991). In the first procedure, a subject has to find the smaller of $Sandstein_1 + Gifte_1$ and *Algen*. First they need to find the value of *Algen* and the first values in the lists of *Sandstein* and *Gifte* to substitute them into the equation. Next they can calculate the sum inside the parenthesis and put the smaller value between it and *Algen* as the result.

## Model

`chunktypes, pre-knowledge`

The model was made using the ACT-R architecture through the pyactr

**Table 3**

*Experiment Procedures.*

| Procedures |
|---|
| $(Sandstein_4 - Sandstein_2) * Mineralien$ |
| $(2 * Algen) + Sandstein_{min}$ |
| $Gifte_{max} + Gifte_{min}$ |
| $(Mineralien * 2) - Gifte_4$ |
| $Das\,Höhere\,von\,(Gifte_3 - Gifte_2), (Sandstein_3)$ |
| $Das\,Kleinere\,von\,(Sandstein_1 + Gifte_1), (Algen)$ |
| $100 - dem\,Höchstem\,aller\,Ergebnisse$ |

*Note.* The seven translated procedures used in this experiment. Six of them are used in the training phase, in the testing phase one procedure is swapped with the unused one. The bottom procedure is always included as it calculates the total water quality.

implementation. It has the subsymbolic system, production compilation and utility learning enabled.

The model works with four different types of chunks specified. Number chunks hold the number, its digits and the number one higher. Math operation chunks hold an operation, two arguments and a result. Procedure chunks hold the operations, variables and values that make up a procedure in the experiment. The math goal chunk is used in the goal buffer and hold various slots used for operations, like the current operation, arguments, counters and flags.

The model gets some basic knowledge that does not have to be learned in the form of chunks set at model initialization. It knows each procedure already and can retrieve its operations and values with an key. It knows all numbers from 0 to 999 through the number chunktype. It has math operation chunks for all greater/less comparisons for numbers between 0 and 10. It has math operation chunks for addition of numbers between 0 and 21.

All trials are generated before the simulation starts and ordered depending on condition. The model uses an environment to simulate a computer screen. Elements are aranged in columns with the values in rows below their column header. Everytime the user inputs an answer or the variables change, the evironment variables are directly updated. User input and trial change is detected from the model trace.

The model works through the tasks with a set of productions, which perform mathematical operations, search the screen, input answers and organize order of operations.

## Greater/Less-than Operation

This pair operations compares two multi-digit numbers and sets the greater/less number as answer. For each digit (hundreds, tens, ones) there is a set of productions comparing that digit of the two numbers. Each production set for a digit requires all higher-significant digits to be equal. That means that the productions comparing the tens can only fire if the hundreds are equal and the productions comparing the ones can only fire if both the hundreds and the tens are equal. The selected production now retrieves a comparison of the two digits from declarative memory. Depending on the result, either number 1 or number 2 will be written into the answer slot.

## Addition Operation

This operation adds two numbers through column-addition. The first production retrieves the sum of the ones digits of the two numbers. The sum is put into the ones digit of the answer. Next it tries to retrieve an addition operation from memory, where ten plus any number equals the previously found sum. If the retrieval fails, the result of the ones addition was less than ten and no carry-over is necessary. If the retrieval succeeds, a carry flag is set and the second addend of the retrieved operation (the part over ten) is set as the ones digit answer. Now the sum of the tens digits of the numbers is retrieved. If the carry flag is set, add one to the sum. Again check for remainder and set a carry flag if necessary. Then the same repeats for the hundreds digits.

**Multiplication Operation**

This operation multiplies two numbers through repeated addition. Multiple productions handle cases in which one of the arguments is one or zero and directly set the answer accordingly. First, it tries to retrieve the sum of the second argument plus itself and sets a counter to one. If the retrieval succeeds, set the answer to the sum and increment the counter by one. While the counter is not equal to argument 1, retrieve the sum of argument 2 plus the result and increment counter. If the counter is equal to argument 1, the operation is finished. If the retrieval of the sum fails, save arguments and counter in different slots and change the current operation to addition, as well as the next operation to multiplication. When the current operation is multiplication again and there are values in the saved argument slots, restore arguments and continue.

**Subtraction Operation**

The subtraction algorithm uses the austrian method, by checking for each digit if the subtrahend is greater than the minuend. If not, it can safely subtract the two digits and move to the next one. If yes, the subtraction will be done after increasing the minuend by 10. Additionally a carry variable will be set, which increases the subtrahend by 1 on the next digit.

**Motor System**

The motor module is used to input the answers and to press continue. When the current operation is to type the answer, the first production requests the tens digit to be pressed on the keyboard. When the action is finished, the ones digit and spacebar to continue are requested to be pressed in turn.

**Visual System**

The visual module is used for various operations to find the current task or to replace variables in a task with the values shown on the screen. The screen is organized in columns with headers, so the visual module first searches for the correct column by keyword. Now different kinds of searches will be performed dependent on what is requested.

To find the next task, the search goes down the column of tasks and saves the task at the current row. If there is nothing in the answer column at the same y-coordinate, the currently saved task was not answered, the search is done. To find a variable value by index, the search travels down the column while counting and stops at the desired index. To find the max/min value of a variable, the search travels down all values in the column, checks for each one if it is greater/less than the currently saved value and replaces it if necessary. Once all values are checked, the search is finished.

**Utility Operations**

Several productions dictate in what order operations are executed. When the operation slots are empty, the visual search for the next unanswered task is started. When a task is found, productions check if the argument are already numbers and if not, request the visual search for substitution with the values on screen. When a task is finished, the result is saved in a slot and other slots are reset, starting task search again. If the second task is finished, start the motor input of the answer.

One production detects if the current operation is finished and another operation is queued, and sets the next operation.

Since operations use both the full numbers and their digits, a set of productions fills digit slots with the digits of a number and vice versa.

**Figure 2**
*Logic Flow of Addition*

*Note.* When each production is executed depending on state. Either example for one operation or figures for all?

## Results

Without enabling the subsymbolic system and its learning algorithms, the average time the model takes to solve a specific procedure stays the same over the experiment (Figure 3). This is expected; while each finished mathematical operation does get remembered by the model, the amount of argument with operation permutations is too high to be useful in this few trials.

Due to multiple roadblocks in working with the subsymbolic system in pyactr, it was not possible not simulate a full experiment run with it enabled. Details about these difficulties will be reviewed in the Discussion.

**Figure 3**
*Mean solution time in training and transfer phase*

*Note.* Mean solution time of all six procedures of a water sample in blocks of five samples.

**Figure 4**
*Comparison with human experiment*

## Discussion

### Working with ACT-R/pyactr

no basic productions given (aside basic tutorial code), everything has to be implemented from scratch, papers using act-r very rarely publish their model code

no/confusing task switching/subgoals

this model uses many different operations and modules of ACT-R and has to model each from scratch and handle task switching

vis: relative positions are not implemented, the visual search loops had to be unrolled to the required number of iterations and is not general

### Model Improvements

More in-depth modeling of all operations

track working memory usage

## References

Elio, R. (1986). Representation of similar well-learned cognitive procedures. *Cognitive Science*, *10*(1), 41–73. https://doi.org/10.1207/s15516709cog1001_2

Frensch, P. A. (1991). Transfer of composed knowledge in a multistep serial task. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *17*(5), 997–1016. https://doi.org/10.1037/0278-7393.17.5.997