Modeling of Transfer in Complex Tasks

Niclas Andreas Dobbertin
Technische Universität Darmstadt

Abstract

A model which simulates learning also has to account for the effect transfer has on new skills. Learning a skill that shares steps with a previously learned one speeds up acquisition. This thesis presents an ACT-R model of a task used by Frensch (1991) to investigate transfer learning. It will give a general overview of learning in production systems and explain the components of the model. Due to bugs in the used ACT-R implementation no results can be presented, however pain points in working with ACT-R will be discussed to motivate future work.

Modeling of Transfer in Complex Tasks

Introduction

When trying to understand how humans learn, transfer learning is particularly interesting. Skills acquired by training can speed up acquisition of different skill through some mechanism. Modeling this mechanism needs to take in account all of the steps the mind goes through when solving a task to re-use, or rather transfer them to another task. Unified Theories of Cognition are what Newell (1994) argues to be the approach to gain a complete understanding of the human mind. Also called cognitive architectures, they combine all of the specialties of the mind into one single framework, that ideally completely mimics what the human mind does. Using such an architecture, it should be possible to describe a task in detail and observe transfer learning to another task.

Transfer learning was previously examined by Frensch (1991) to differentiate the transfer effects between learning the components of a task and learning the composition of components in a task. They used an experiment shown by Elio (1986), which uses multi-step mathematical equations, which have to be learned in different ordering conditions. To test transfer, one equation is swapped to a new one and the speed of learning it is taken. This kind of task seems appropriate to model in a cognitive architecture to see how it predicts transfer learning.

ACT-R (Anderson et al., 2004) is an established cognitive architecture that uses productions to model procedures in the mind. There are several methods that use these productions to describe learning. Brasoveanu and Dotlačil (2021) compared different reinforcement learning algorithms in one such method, although using a lexical task. For this they created a re-implementation of ACT-R in python (Brasoveanu & Dotlačil, 2020), which seems like a good starting point to implement Elios task in a model.

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.

Learning

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).

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.

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 lime 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

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.

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 acquisition 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 transfer trials take place, in which the third procedure from the acquisition phase is switched for the unpicked one. There are three conditions that determine the order in which procedures are presented in the acquisition 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 transfer phase always uses fixed order.

How modeled:

(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

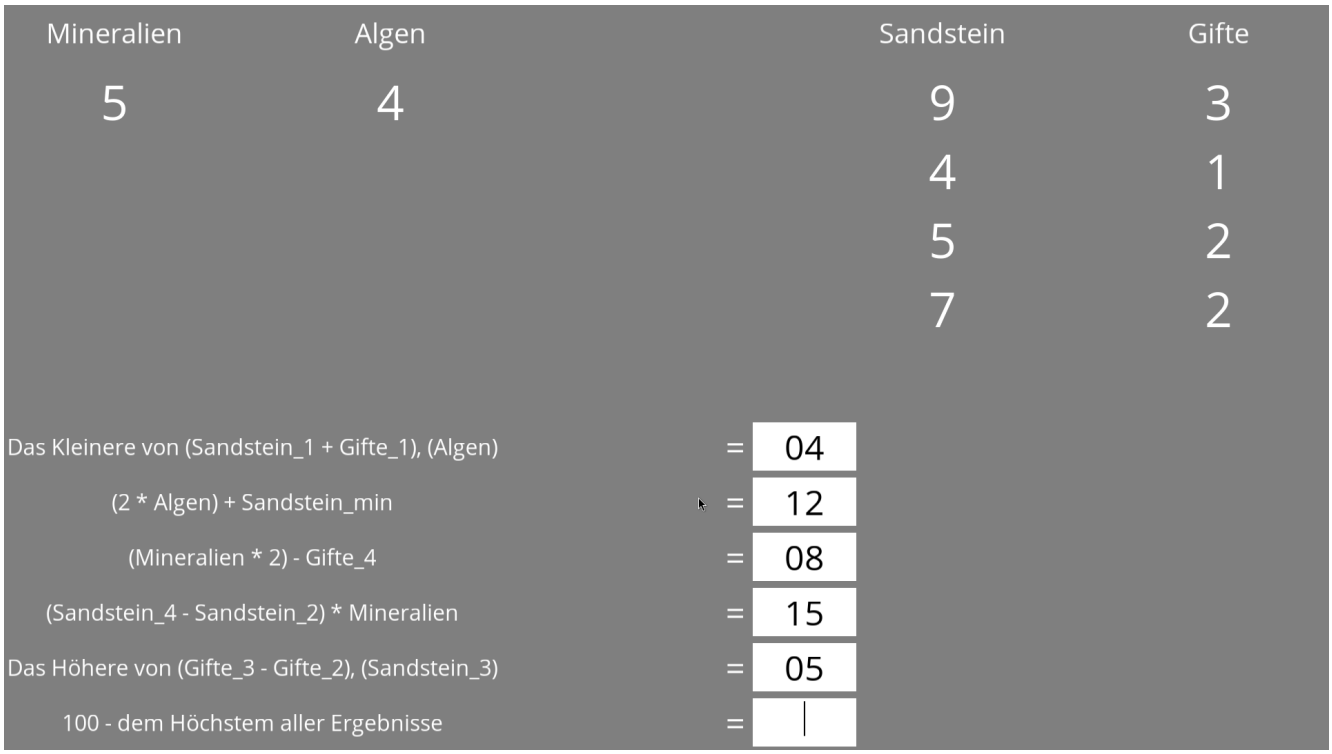
(c) *A Compiled Production*

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.

Figure 1
Screenshot of experiment display



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

The goal of the model is an accurate representation of how a human would solve this task and improve over time. Optimally the models solving time would, in each

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 acquisition phase, in the transfer phase one procedure is swapped with the unused one. The bottom procedure is always included as it calculates the total water quality.

condition, improve similarly to previous human results. Looking at the ways an ACT-R model can improve, production compilation seems to be the important function compared to utility or chunk learning. A lot of small subtasks have to be accomplished for a single trial, such as finding correct variable values, solving multiple mathematical operations and typing the answer. These steps however need to be repeated for each trial and while the numbers and with them the mathematical operations can change a bit, the overall order and structure of subtasks stays the same. Production compilation therefore promises strong improvements to solving times, as many steps can be combined into a single one, eliminating time deciding on the next step. Additionally numbers in this task are often small, allowing some common operation to be saved as productions in procedural memory, removing time calculating or trying to retrieve from declarative memory.

Utility learning is needed to evaluate the usefulness of compiled productions, but since the task and subtask order is very rigid, should have no important role in learning otherwise. Chunk learning doesn't seem impactful either, as there are too many permutations of variable values and too few trials to memorize helpful information.

To complete the experiment in a manner a human adult would, the model is given a baseline of knowledge and skill to start with. This includes basic knowledge of possible numbers and mathematical operations it has to solve.

Implementation

The model was made using the ACT-R architecture (Anderson et al., 2004) through the pyactr (Brasoveanu & Dotlačil, 2020) implementation. The base model uses default parameters. To enable production compilation and utility learning, the parameters "production_compilation" and "utility_learning" have to be set to "True". Due to implementation details in pyactr, the subsymbolic system has to be enabled as well. Issues and workarounds when implementing the model will be reviewed in the Discussion.

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 still has to find the right key by visually searching for the current procedure on the screen 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 20. It has math operation chunks for addition of numbers between 0 and 20.

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 arranged in columns with the values in rows below their column header. Every time the

user inputs an answer or the variables change, the environment 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 10 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 10) 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 1 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 1 or 0 and directly set the answer accordingly. First, it tries to retrieve the sum of the second argument plus itself and sets a counter to 1. If the retrieval succeeds, set the answer to the sum and increment the counter by 1. 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 space bar 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.

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. 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.

Discussion

This model shows that it possible to implement the task in ACT-R and that it should be possible for the model to produce task solving time for comparison with human subjects. During development however, a variety of difficulties emerged and

ultimately prevented ACT-Rs learning functions to simulate human learning. An ACT-R implementation in python, pyactr, was used to program this models, which brought some pyactr-specific problems with it. Difficulties in re-implementing ACT-R were already mentioned in Albrecht et al. (2014), who state that today ACT-R is specified by its implementation, rather than a formal specification. The implementation of production compilation in pyactr seems to include some critical bugs, causing the model to crash when compiling some productions. While it showed that production compilation works in most cases, this stops it from being utilized in a model and prevented us from investigating its effect in our task. Another problem was the missing implementation of relative coordinates in visual search, meaning scanning objects left to right for a specific one is not possible and had to be circumvented by hard-coding all possible object positions to search. Since it is developed by very few people, it is sadly natural that specific parts and usages are not working correctly, despite being functional in general.

In general, it was not clear how a production or set of productions has to be written in order to achieve some task correctly. While ACT-R gives a lot of tools to handle many situations, it was surprising that even even basic operations like multi-digit addition or subtraction do not have an example implementation. To use the goal buffer or the imaginal buffer, how to sequence tasks, how general or specific should productions be and how much strict order should the goal buffer enforce were important considerations during development and had to be answered more by feeling than by knowledge from references. There are various models used in ACT-R tutorials to introduce its capabilities, these however are very limited and don't expand beyond very simple tasks. Papers usually don't include the exact model and productions used, which leaves few examples and general guidelines to new model makers. Implementing new models would be much easier, if something similar to a software library exists for ACT-R. It could contain simple, common tasks like for example mathematical and lexical operations, visual search and handling task switching or subgoals. Such a library would additionally serve as an example of proper implementation of different productions in ACT-R, giving guidelines to newer model makers.

Model Improvements

Most importantly, solving the production compilation problem and actually comparing the models learning behavior with human data would be the next step from this point on.

While model currently does not work correctly, there are a variety of improvements possible after technical issues are removed. Mathematical operations could be modeled much more general and to work with higher and negative numbers. This would make it possible to learn mathematical facts from the ground up, instead of relying on a set of given knowledge. Introducing multiple ways of doing an operation, like addition by counting from 1 or from the first argument, as well as shortcuts like swapping arguments in applicable operations, gives the model opportunity to utilize its utility learning more. Another important improvement would be better switching between tasks, as e.g. multiplication requires additions being performed. This required

a complex set of production, which a general task switching implementation could simplify.

It would be interesting how other cognitive architectures behave in comparison to ACT-R. SOAR (Laird, 2022) and especially PRIMs architecture (Taatgen, 2013), which specializes in transfer of knowledge through small knowledge bits.

References

Albrecht, R., Giesswein, M., & Westphal, B. (2014). Towards formally founded act-r simulation and analysis. *Cognitive Processing*, *15*, S27–S28.

Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S., Lebiere, C., & Qin, Y. (2004). An integrated theory of the mind. *Psychological review*, *111 4*, 1036–60. https://api.semanticscholar.org/CorpusID:186640

Brasoveanu, A., & Dotlačil, J. (2020). The ACT-r cognitive architecture and its pyactr implementation. In *Language, cognition, and mind* (pp. 7–37). Springer International Publishing. https://doi.org/10.1007/978-3-030-31846-8_2

Brasoveanu, A., & Dotlačil, J. (2021). Reinforcement learning for production-based cognitive models. *Topics in Cognitive Science*, *13*(3), 467–487. https://doi.org/10.1111/tops.12546

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

Laird, J. E. (2022). Introduction to soar. https://arxiv.org/abs/2205.03854

Newell, A. (1994). *Unified theories of cognition*. Harvard University Press. https://books.google.de/books?id=1lbY14DmV2cC

Taatgen, N. A. (2013). The nature and transfer of cognitive skills. *Psychological Review*, *120*(3), 439–471. https://doi.org/10.1037/a0033138