Simulating Learning From Language and Examples

Daniel Weitekamp[1], Napol Rachatasumrit[1], Rachael Wei[2], Erik Harpstead[1], and Kenneth Koedinger[1]

[1] Carnegie Mellon University, Pittsburgh PA 15289, USA
[2] University of Illinois Urbana-Champaign, Champaign IL 61820, USA

Abstract. Simulations of human learning can be used as computational models for evaluating theories of learning. They can also be taught interactively to author intelligent tutoring systems. Prior simulated learner systems have learned inductively from worked examples and correctness feedback. This work introduces a mechanism where simulated learners can also learn from natural language. Using a neural grammar parser with additional symbolic processing steps, we simulate the production of loose interpretations of verbal instructions. These interpretations can be combined with worked examples to resolve the ambiguities of either form of instruction alone. We find that our system has practical benefits over an alternative method using github Copilot and slightly better accuracy.

Keywords: Learning Simulations · Computational Models of Learning · Natural Language Processing · Code Synthesis · Authoring Tools

1 Introduction

A computational model of learning is a student model that learns directly from instructional material, as humans do, and produces particular incorrect and correct responses as a result. By contrast statistical models of learning are fit to student data and typically reduce the cognitive complexities of learning to numerical predictions of performance [2]. Computational models of learning embody executable theories of learning that can be applied to learning materials to evaluate instruction *a priori* without fitting to student data [8]. While a statistical model may characterize learning as a change in the probability of correct responses, a computational model produces particular responses to question items and has interpretable underlying reasons for them.

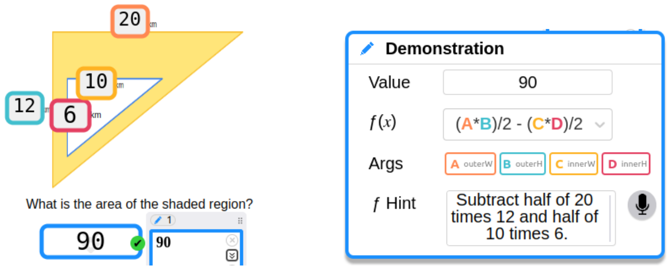
Prior simulated student technologies used as computational models of learning have predominantly learned from worked examples and correctness feedback [5][6]. This work expands the set of instructional modalities that simulated students can learn from to include natural language instruction.

Worked examples alone can be ambiguous. The particular content of a worked example often lends itself to many different explanations. For instance for the problem:

```
Find the slope of the line that passes through (5, 4) and (7, 8)
```

The worked example solution is 2, and the correct explanation in this case is $(8-4)/(7-5) = 2$. However, there are also several incorrect explanations like $8/4 = 2$ or $7-5 = 2$. Prior simulated learners have used an error-prone brute-force guess-and-check style method called *how-learning* to induce these sets of operations. This work aims to allow simulated learners to learn more robustly by interpreting natural language instruction to disambiguate the operations used to produce worked examples.

Simulated students can also be used to rapidly author intelligent tutoring systems (ITS). ITS authoring can be a time consuming process. For instance, programming an hour of cognitive tutor based instruction takes about 200-300 hours of development time [1]. ITS authoring with simulated learners has been shown to be considerably faster even than GUI-based authoring methods [7]. Yet prior simulated learner based authoring systems have suffered from induction errors that put domain-general authoring out of reach. Typically simulated-learner's *how-learning* mechanisms help authors to program formulae that grade and producing bottom-out hint answers for problem steps. Enabling this functionality for domain-general authoring presents a challenge: when a large set of functions are made available to *how-learning*—which might include many kinds of functions beyond just arithmetic operations—*how-learning* may search through an intractably large space of function compositions, making it prone to stopping short on incorrect formulae that reproduce worked examples but that are incorrect in general (like in the example above). The method we present in this work allows authors to verbally clarify the composition of operations used in their worked examples. Our system interprets natural language and examples together to overcome the intractability of performing *how-learning* on demonstrated examples alone.



What is the area of the shaded region?

90

Demonstration

Value	90
$f(x)$	(A*B)/2 - (C*D)/2
Args	A outerW, B outerH, C innerW, D innerH
f Hint	Subtract half of 20 times 12 and half of 10 times 6.

Fig. 1: (left) A worked example "90" for a triangle area composition problem. (right) the user's recorded spoken instruction "f Hint" and the induced formuale "f(x)".

2 Related Work

The majority of prior work on natural language processing of mathematical language pertains to the translation of word problems directly into equations [9]. By contrast, our system interprets operational langauge-based instruction, and outputs executable knowledge structures that can perform steps in mathematical procedures. Large language models (LLMs) like OpenAI's Codex [3] are closer to our method in terms of functionality, in the sense that, like our system, they can synthesize executable formulae. We compare our system to Codex via the Github CoPilot plugin.

3 Better Models of Novices than LLMs with Neuro-Symbolic AI

Simulations of learning must make theoretical commitments about both learner's prior knowledge and about the content of the target knowledge being taught. The goal of a computational model of learning is to explain how knowledge is constructed and refined through learning. LLMs don't hold much promise of helping with these sorts of simulations because insofar as they exhibit mastery of capabilities that students typically learn, they have acquired those capabilities from many domain-specific experiences greater in number and diversity than any human would ever encounter in a lifetime (like millions of github repositories for Codex). While LLMs boast impressive generative capabilities, their learning process is considerably less data-efficient than human learning. Additionally their knowledge is largely encoded in unexplainable "blackbox" weights, acquired from often proprietary datasets. Most of all, LLMs simply know too much to be useful for modeling learning. A pre-trained model that already possess the domain-specific capabilities that one intends to simulate the acquisition of is useless for modeling a novices' learning trajectory from first experiences to mastery.

By contrast, our approach restricts itself to only the use of a pre-trained neural grammar parser [4] and coreference resolver[3], but uses no text generation models. Our use of a pre-trained grammer parser assumes that our simulated students can, as prior knowledge, parse the structure of English sentences, but does not assume an ability to translate mathematical language to executable operations or written equations. Our system transforms grammatically parsed sentences in several hard-coded yet domain-general processing steps to produce search policies for guiding a typical simulated learner's *how-learning* mechanism. These policies embody loose interpretations of sentences, which enclose small spaces of possible function compositions intended by the input sentence. Combining these policies with the typical search process used in *how-learning* disambiguates the formulae an instructor or ITS author intended to teach with their combined worked examples and natural language instruction.

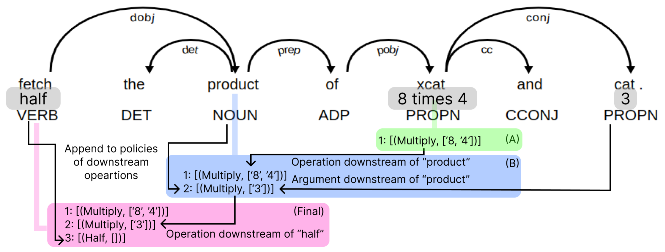

dobj
det
prep
pobj
cc
conj
fetch
half
VERB
the
DET
product
NOUN
of
ADP
xcat
8 times 4
PROPN
and
CCONJ
cat .
3
PROPN
1: [(Multiply, ['8, '4'])]
(A)
Append to policies of downstream opeartions
Operation downstream of "product"
(B)
1: [(Multiply, ['8', '4'])]
2: [(Multiply, ['3'])]
Argument downstream of "product"
1: [(Multiply, ['8', '4'])]
2: [(Multiply, ['3'])]
3: [(Half, [])]
(Final)
Operation downstream of "half"

Fig. 2: Parsing and policy for "half the product of 8 times 4 and 3". Boxes (A), (B), and (Final) show the recursive process of a policy being constructed by traversing the grammatical parse of a sentence.

For authoring purposes, we don't need to constrain prior knowledge of simulated learners. Thus, if an LLM like Codex proves better than our approach at producing target formulae—even if only because somewhere in its vast training set there is an example analogous to the target task—then it may ultimately be the preferable tool. Yet, authoring still requires domain-general generative capabilities, so if Codex is relying on instances from its training set, in lieu of broader generative capabilities, then it may fail when tasked with aiding authors at building one-of-a-kind materials. We evaluate this possibility by comparing our system with Codex on descriptions of made-up formulae.

[3] https://spacy.io/universe/project/coreferee

4 Methods

We recruited 10 crowd workers through Prolific to generate natural language instructions (i.e. hints) for problem steps. Participants solved 14 unique math problem steps, and provided conceptual hints (i.e. "describe the concept in broad terms") and grounded operational hints, which we requested include all operations and values used to produce the answer, without using mathematical notation—they should be written as if spoken aloud. Conceptual hints were requested simply to highlight the requirements of operational hints and are not used in our evaluations. Participants were given a five question *check your understanding* survey to ensure they understood these distinctions.

Two of the authors independently coded each participants' grounded operational hints to mark if they were indeed both grounded (i.e. "mentions all required arguments and constants"), and operational (i.e. "mentions all required operations"). An inter-rater reliability of 97.2% was achieved, and the discrepancies were resolved through discussion. Hints marked as both grounded and operational, we refer to as *good* hints.

For each response where participants produced the correct answer, we ran the participant's hints and worked examples through our system, and also with worked examples only. We did the same with Codex via Github Copilot. In this case worked examples, which were used to eliminate functions based on return value. We also ran Copilot with hints only. Our system had 7 functions "Add, Multiply, Subtract, Divide, Half, OnesDigit, and Square" available to it, which were sufficient for building function compositions for all of the target formulae, plus 8 additional functions not needed for any of the target tasks "TensDigit, Power, Double, Increment, Decrement, Log2, Sin, and Cos".

Typically Github Copilot produces a function implementation from a function header and doc-string. We filled our participants' grounded operational hints into Github Copilot as the doc-string of an empty Python function with the header `foo():`, and recorded the extended set of suggestions. Similar to our system's output this constituted a small variable set of candidate solutions. Typically Copilot uses arguments specified in the function header. This information is not present in the participant's hints, so we omit arguments and just evaluate whether Copilot suggested implementations that were functionally equivalent to the target formulae, expressed with constants instead of arguments. When examples were included, we also executed each of these functions to see if they reproduced the worked example.

For both systems we measured whether or not a correct formula was produced for each grounded operational hint, and counted the number of incorrect formulae produced. We considered any algebraic rearrangements (e.g. $A * B = B * A$) of the target formula to be correct. Thus, we use the average number of incorrect formulae as the principle measure of error magnitude, instead of proportion correct, which would be sensitive to returning several isomorphic formulae. The principal measures of the rate of correctness are (1) the percentage of the participant provided hints where the system produces at least one correct formula (i.e. *has correct*), and (2) the percentage where only the correct formula is produced (i.e. *only correct*).

To evaluate Codex's potential performance on one-of-a-kind formulae unlikely to be present in its training set, we repeated these evaluations for a set of 10 made-up formulae with accompanying grounded operational hints that we wrote ourselves.

5 Results

Each of our 10 participants finished 14 problems for a total of 140 responses. We removed 26 responses where participants produced incorrect answers and used the remaining 114 grounded operational hints for evaluation. Our system produced sets of formulae containing at least one correct formula 82.4% of the time, and 69.2% of the time only correct formulae were returned. Of the 114 grounded operational hints 87 were coded as *good*. On average our system performed better for the *good* hints, 86.2% had correct formulae and 73.6% had only correct formulae. For the set of all 114 responses our system produced an average of 1.54 unique incorrect formulae, whereas for the *good* hints it produced an average of 0.54 incorrect formulae.

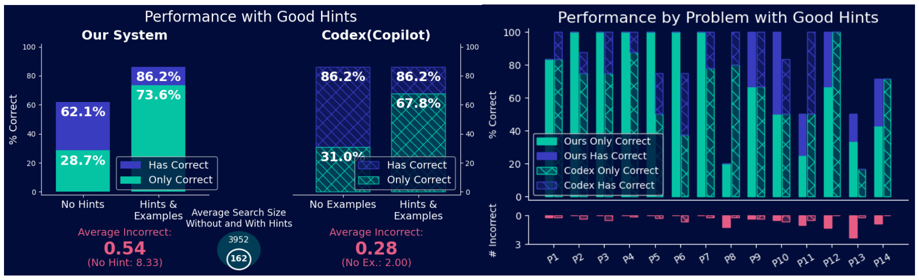


Fig. 3: Overall performance and performance by problem on *good* hints for our system and Github Copilot. Reduction in *how-learning* search size from hints is shown.

Overall our system's performance (Figure 3) was improved considerably by parsing hints in addition to worked examples. When using only worked examples without hints, as in prior work, our system produced only correct solutions 28.7% of the time. Succeeding on just 4 of the 14 problems. Hints also reduced the average number of incorrect formulae from 8.33 to 0.54, and the average search space size of *how-learning* from 3952 function compositions to 162.

Github Copilot achieved the same *has correct* performance as our system on *good* hints with a slightly worse *only correct* performance of 67.4%. Our system performed nearly perfectly on problems 1-7 which had shorter formulae and participant hints—an average of 16 words, versus 24 words in problems 8-14. Most of our systems' errors came from difficulties parsing the verbose hints from problems 8-14. Copilot's performance was more varied, but was much higher than our system for some problems (e.g. P8, P11). This is likely partially due to mimicking of similar problems in Codex's training set, since in some cases it produced code with suspiciously domain-specific inline comments and variable names. However, Copilot also performed perfectly on our corpus of 10 made-up hints and worked examples, verifying that it is indeed strong at novel code generation. By contrast, our system also produced correct formulae for all 10 made-up hints, and only a few incorrect formulae for 4 of them.

6 Discussion

Both systems showed improvements consistent with the assertion that the combination of langauage and worked examples benefits instruction comprehension over either taken alone. Our system generally performed best on more concise hints—a pattern of performance that bodes well for computational modeling purposes. Humans often learn better from concise directed learning experiences. Our system performed most poorly on problems like problem P8 (Figure 1), that are often scaffolded into multiple steps in ITS interfaces. Copilot by contrast had no consistent performance pattern, and showed signs of leveraging prior domain-specific knowledge.

7 Conclusion

Future computational modeling work may investigate how these language comprehension abilities compare to human capabilities. In this work we've demonstrated two means of generating knowledge from grounded operational instruction. This is however only a first step. Our system takes a relatively structured approach, compared to the method using Copilot, making it conducive to many future refinements and investigations. This opens opportunities to investigate questions of how learners interpret tutorial instruction, and how instruction and ITS hints may be improved as a result.

For ITS authoring purposes our crowd-worker results may understate the efficacy of our approach. For instance, authors may get better at generating hints over time, or use the experience as a starting point for authoring their own formulae directly—something that is not obvious, especially if non-arithmetic functions are involved. Additionally, the fairly small average number of incorrect formulae produced by both systems means correct formulae can be easily selected from among a small set of candidates. Overall, our method bodes well for incorperating natural language processing into ITS authoring and computational models of student learning.

References

1. Aleven, V., McLaren, B.M., Sewall, J., Van Velsen, M., Popescu, O., Demi, S., Ringenberg, M., Koedinger, K.R.: Example-tracing tutors: Intelligent tutor development for non-programmers. International Journal of Artificial Intelligence in Education **26**(1), 224–269 (2016)
2. Cen, H., Koedinger, K., Junker, B.: Learning factors analysis–a general method for cognitive model evaluation and improvement. In: International Conference on Intelligent Tutoring Systems. pp. 164–175. Springer (2006)
3. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.d.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al.: Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021)
4. Honnibal, M., Montani, I.: spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing (2017), to appear
5. MacLellan, C.J., Harpstead, E., Marinier III, R.P., Koedinger, K.R.: A framework for natural cognitive system training interactions. Advances in Cognitive Systems **6**, 1–16 (2018)
6. Matsuda, N., Cohen, W.W., Koedinger, K.R.: Teaching the teacher: Tutoring simstudent leads to more effective cognitive tutor authoring. International Journal of Artificial Intelligence in Education **25**(1), 1–34 (2015)
7. Weitekamp, D., Harpstead, E., Koedinger, K.: An interaction design for machine teaching to develop ai tutors. CHI (2020)
8. Weitekamp, D., Harpstead, E., MacLellan, C.J., Rachatasumrit, N., Koedinger, K.R.: Toward near zero-parameter prediction using a computational model of student learning. International Educational Data Mining Society (2019)
9. Zou, Y., Lu, W.: Text2math: End-to-end parsing text into math expressions. arXiv preprint arXiv:1910.06571 (2019)