Inferring Pictorial Transformations by Program Induction

Max Krieger[1], Kevin Ellis[2]
[1]Needham High School
[2]Massachusetts Institute of Technology

Humans are capable of inferring visual rules and patterns given few examples of their use. Likewise, they can express new rules by providing examples alone, and expect others to easily understand them. The ability to observe, infer, and express visual rules using a limited set of examples may be trivial for people, but there are few computational analogues. We present a system that learns and produces image transformation rules by synthesizing programs. We designed a means of observing features in examples, a model that learns to produce likely rules, and a language to express hypothesized rules. Trained on 12 annotated problems, the system has an accuracy rate of approximately 70% when tasked with solving 20 novel problems, and performs approximately 16 times faster than an untrained baseline. The project combines the disciplines of machine learning and programming languages, and has relevance to both computational cognitive science and human-computer interaction.

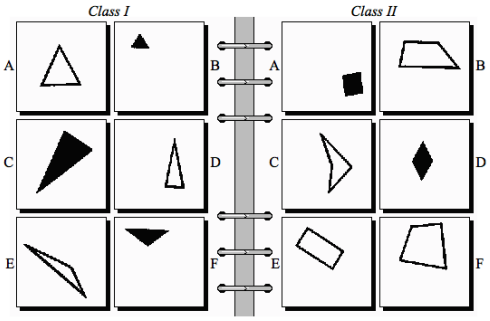

Figure 1: What property applies to the left that doesn't apply to the right? (Foundalis)

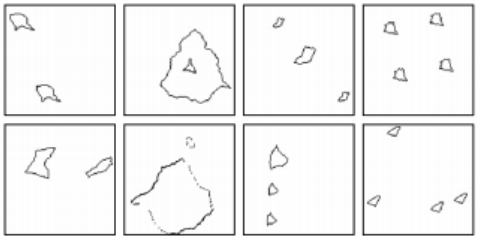

Figure 2: What is the rule difference between the top and bottom scene for each column? (Fleuret et al.)

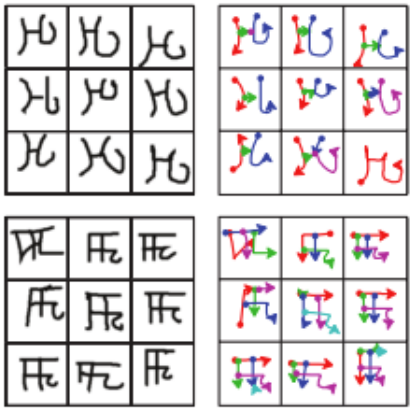

Figure 3: What strokes might a human make to draw a symbol? (Lake et al.)

1 Introduction

For humans, finding patterns in images comes intuitively. With examples alone, we can learn to handwrite symbols (Lake et al.), categorize complex scenes (Fleuret et al.), and extrapolate transformations with little effort and time (see figures 1, 2, 3). In general, humans have been shown to be capable of deriving strong *hypotheses*, or common rules, for example-based *problems*. Even on the order of one or two examples, humans can find hypotheses that predict how transformations, compositions, or categorizations occur when given additional unsolved examples (Lake et al.).

Though these problems may be trivial for humans to solve, inferring hypotheses is diffi-

cult to approach computationally. Humans are highly capable of both *observing* the features that stand out in examples, and *expressing* hypotheses in language and thought that can be remembered and executed. To emulate these capabilities, computers often rely on the concept of exploring a problem *search space*, or the set of possible solutions, using a heuristic method. However, search spaces for example-based problems can be enormous, especially for richer problems such as inferring textual or graphical transformations.

The problem becomes even more difficult when one considers that an adult human might only need 2-3 examples to learn a distinct concept or pattern. Counter-intuitively, using a computational learning model might require corpora of hundreds to hundreds of thousands of examples to be on par with the human's accuracy (Lin et al.). The domain of visual learning by example, in particular, becomes not only a challenge of engineering, but a challenge of intelligence.

What use is it to teach a computer to learn rules by example? In contexts involving human-computer interaction, human-provided examples can often close the lingual gap of describing procedures to a computer. For instance, suppose a user wanted to extract a list of first names from a spreadsheet containing a list of strings of first, middle, and last names. Most users are not literate in Regular Expressions, let alone any programming language that a computer would understand. Likewise, computers struggle to interpret the English procedure "Put the first names into a new list" without context and sophisticated lexical ability. A potential compromise of communication between human and computer would be for the user to provide a small set of input-output examples. From this information, the computer could infer a concrete hypothesis of the user's desire in the form of a familiar executable procedure: a *program*.

Much has been done to explore how programming-by-example can be applied to spreadsheet (Gulwani) and text manipulation (Balog et al.). The approach has also been used to teach computers high level human concepts such as handwriting (Lake et al.), list manipulation algo-

rithms (Ellis et al.), and even physical intuitions (Lake et al.). However, little has been explored in the vast domain of image manipulation, producing software utilized daily by hundreds of thousands of people (United States Department of Labor). Teaching computers human-level image-based pattern and transformation skills would not only reveal new, efficient ways for computers to learn concepts, but would also greatly improve how we interact with computer graphics. Giving image editing software the capability of inferring the user's intentions could streamline various tasks that would otherwise be complex or menial.

We propose a system that can hypothesize on graphics transformations by synthesizing programs. The problem domain that the system can solve is based on a set of *transformations* that occur on entities over simple 2-dimensional *scenes*. The system is capable of observing visual properties of scenes, and using the observations to find and express likely hypotheses. We teach a machine learning model to find a relationship between observation and hypothesis, and design a language to express hypotheses in an executable, reproducible form.

2 Approach

2.1 Representing a Scene

I chose to represent a scene in data as a set of *Shapes* on a 2-dimensional x, y plane. Shapes have attributes of *Color* ($\{Red, Blue\}$), *Geometry* ($\{Circle, Square\}$), x position ($\mathbb{Z}_{\geq 0}$), and y position ($\mathbb{Z}_{\geq 0}$). These attributes open a large enough problem space that a variety of graphical patterns and transformations can be expressed. A *transformation* is thus defined as a pair of scenes, and a *problem* is a set of transformations. A hypothesis for a problem is considered *valid* if it, when applied to the first scene in a transformation, will result in a scene equivalent to the second scene in a transformation.

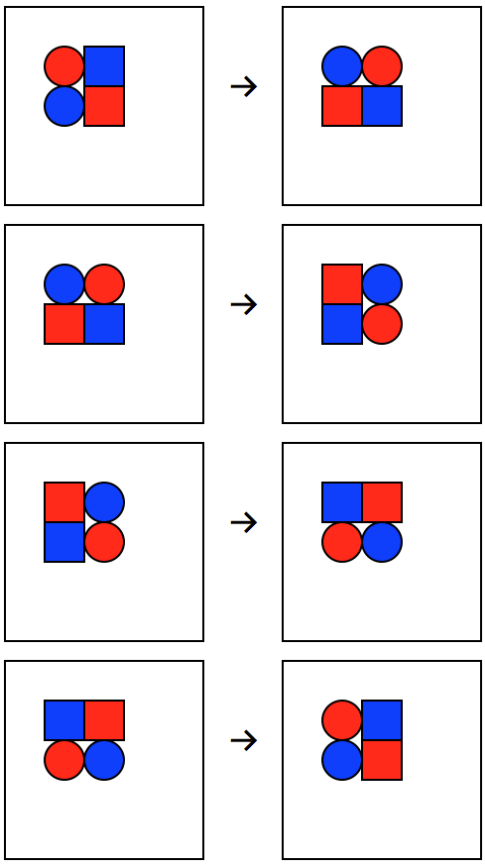

Figure 4: An example of our problem domain. What rule might apply to all transformations?

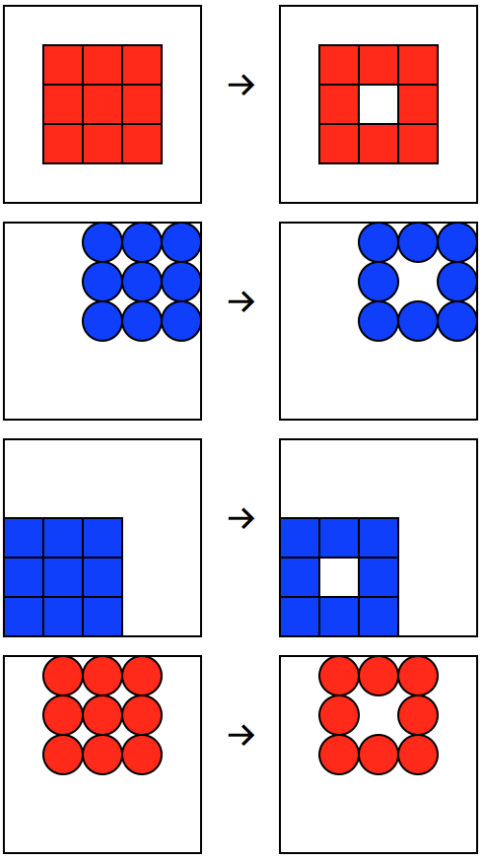

Figure 5: Regardless of unique attributes between scenes, it is trivial to find a common rule.

2.2 Feature Extraction

To be able to reason about different problems, the system must be capable of observing distinct *features* within them. A feature is an extracted scalar property of a set of transformations. Because scenes are represented as simple data structures, making human-like observations on their appearance often only involves sampling the nature of concrete attributes. Qualitative observations a human may make include the overall shape, color, and variation of shapes and their groups, and how such properties change when a transformation is run. Many of these can be captured quantitatively with a basic feature set Φ applied to a problem, some members of which are tabulated in table 1. Features ($n = 6$) are then scaled using standardization.

Feature	Description
ϕ_1	Change in mean x position across the transformation.
ϕ_2	Change in mean y position across the transformation.
ϕ_3	Mean unique *Shapes* in attributes *Geometry* and *Color*.
$\phi_{...}$	...

Table 1: A truncated list of some features used in Φ.

2.3 Representing Hypotheses

Hypotheses are represented as *programs* that are expressed using a Context-Free Grammar containing approximately 30 primitive functions (table 2). The grammar provides a means for the system to express transformations that map, differentiate, select, and modify scenes in a highly composable way. Given an expressive grammar, the system can build programs that model a wide range of hypotheses.

To achieve composability, programs are structured into a tree guided by four *combinators* based on the Lambda Calculus: *Primitives*, *Applications*, *Abstractions*, and *Indices*. These combinators direct the execution flow for a program in a consistently reducible way (see figure 6).

Primitive	Description
(map_scene *scene* λ*shape*.*shape*)	Maps over a scene with a function.
(union *scene scene*)	The $\cup$ of two scenes.
(head *scene*)	The first shape in a scene set.
(select_by_vector *scene vector*)	A scene of all shapes at the head of a vector.
(difference *scene scene*)	The set $\setminus$ of two scenes.
(translate *shape vector*)	Translates a shape across space.
(get_color *shape*)	The color of a shape.
(vec_1_1)	A vector $\begin{bmatrix}1 & 1\end{bmatrix}$ pointing $45°$.
(Red)	The color red.
...	...

Table 2: A truncated list of some primitives in the grammar used.

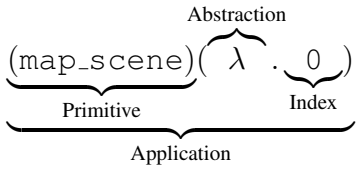

Figure 6: An illustrated example of the Lambda Calculus.

A Primitive simply contains a pointer to a member of the grammar. An Application contains a Primitive applied to a second combinator as a function composition. Because some members of the grammar are higher-order functions (HOFs), Abstractions and Indices provide an elegant solution to the problem of variable binding. An Abstraction introduces a bound variable from the parent HOF into the program *context*. In the traditional Lambda Calculus, a bound variable must be given a unique symbol (typically $\{x, y, z, \ldots\}$), however approaches such as Combinatory Logic and De Bruijn indices have shown that symbols can be eliminated using simpler alternative structures (Pierce). An Index is a pointer to a bound variable in the context represented as a *De Bruijn index*: an integer denoting the nth abstraction from the Index's location in the program tree.

To guarantee valid, well-typed programs, we assigned *types* to all members of the grammar (table 3). This ensures type-sensitive primitives within a program will be applied appropriately,

for example, function that requires a Color as input, will never be given a Geometry.

Type	Rewrites to
$Color$	$Red \mid Blue$
$Geometry$	$Square \mid Circle$
$Shape$	$\{x \in \mathbb{Z}_{\geq 0}, y \in \mathbb{Z}_{\geq 0}, Color, Geometry\}$
$Scene$	$List\{Shape\}$
λ	$(\underbrace{List\{\tau\}}_{\text{arguments}}, \underbrace{\tau}_{\text{returns}})$

Table 3: The types $\tau \in \mathcal{T}$.

2.4 Learning Hypotheses

To predict likely hypotheses from the features of a problem, we created a model that predicts the most probable constituent primitives that will be used. The model learns a probability distribution over the primitives of every type in the set of types $\mathcal{T}$, and assigns this distribution to a *Probabilistic Context-Free Grammar* (PCFG) $\mathcal{G}$ containing all primitives in the grammar and their respective probabilities. At a high level, this approach is similar to other program synthesis schemes which learn distributions over a grammar, seen in Menon et al. and Balog et al.

Suppose then that a relationship between the set of features Φ and the distribution across $\mathcal{G}$ must be found. Such a relationship can be parametrized with a matrix of parameters $\Theta \in \mathbb{R}^{|\mathcal{G}| \times |\Phi|}$, which is learned after training. The probability of an individual primitive $f \in \mathcal{G}$ is thus defined as

$$\tilde{P}(f \mid \Phi; \Theta) = \exp\left(-(\Theta\Phi)_f\right)$$

However, the probability distribution over $\mathcal{G}$ has not been normalized, i.e.

$$\sum_{f \in \mathcal{G}} \tilde{P}(f) \neq 1$$

To normalize all $\tilde{P}(f) \in \mathcal{G}$, we compute a normalizing constant Z for all primitives of type $\tau \in \mathcal{T}$ where

$$Z(\tau) = \sum_{f \in \mathcal{G}_\tau} \tilde{P}(f)$$

A new, normalized probability function is represented as a *cost*:

$$\text{Cost}(f) = -\log \frac{\tilde{P}(f)}{Z(\tau)}$$

What results is a distribution not of fractional probabilities $0 < P(f) < 1$, but of *minimum description lengths* $0 < P(f) < \infty$. Minimum description lengths are an information-theoretic way to effectively represent favorable programs and primitives (MacKay). We refer to the description lengths in $\mathcal{G}$ as *costs*, where for every $f \in \mathcal{G}$, there exists a cost $c \in \mathbb{R}_{\geq 0}$. The cost of a program p is simply the cost of its constituent primitives, or

$$\text{Cost}_{\text{prog}}(p) = \sum_{f \in p} \text{Cost}(f)$$

A greater cost primitive means a less probable one, and thus a greater cost program means a less probable hypothesis.

To learn Θ, the model is trained on a corpus of problems annotated by their respective features and ideal "solution" programs. Finding the best parameters means maximizing the probability of the programs in the corpus when presented with relevant features. This in turn means minimizing the total cost of all programs in the corpus $\mathcal{K}$

$$\arg\min_{\Theta} \sum_{(p,\Phi) \in \mathcal{K}} \text{Cost}_{\text{prog}}(p \mid \Phi; \Theta)$$

I used gradient descent on an automatically-differentiated cost function to approximate an ideal Θ. Because a relationship between $\Theta\Phi$ and $\mathcal{G}$ has been found, the learned Θ can now be used to solve novel problems.

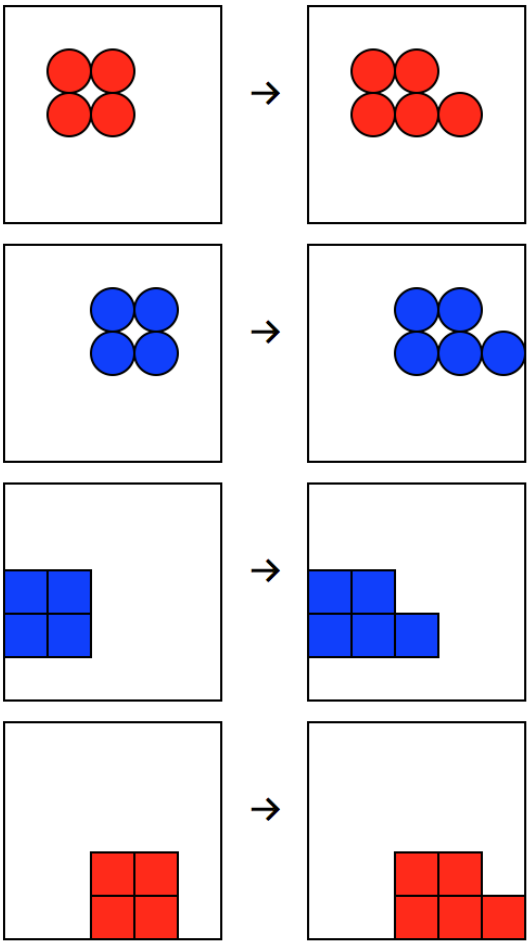

Figure 7: A problem.

```
(union
    ((map_scene
        ((select_by_vector
           $0)
            vec_-1_-1))
        (lambda
            ((translate $0)
               vec_1_0))
    )
) $0
```

Figure 8: A hypothesis inferred by the system conditioned on the problem in figure 7.

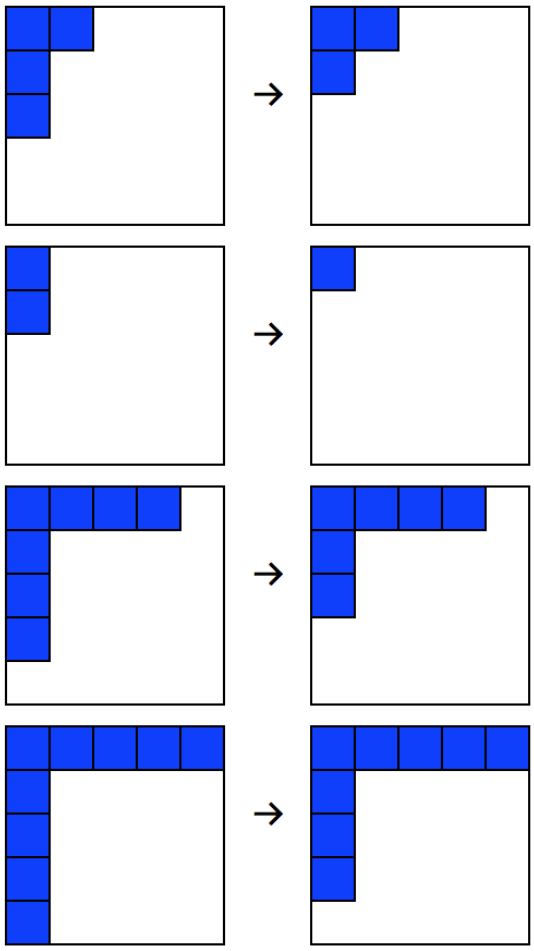

Figure 9: A problem.

```
(difference $0) (
    (select_by_vector $0)
       vec_0_-1
)
```

Figure 10: A hypothesis inferred by the system conditioned on the problem in figure 9.

2.5 Enumerating Hypotheses

Given a PCFG with a distribution of probabilities conditioned on the features of a problem, a hypothesis *enumerator* now generates the most likely programs first. A recursive iterative deepening search (figure 11) searches the space of programs to build candidate program trees, the bounds of which increase until a *cost budget* b depletes. The cost budget penalizes complex, unlikely programs and rewards cheap, simple programs by providing a constraint on the program cost during the search.

On each iteration, the enumerator produces a list of candidate program trees that are type-equivalent to the desired return type of the program. If the candidate primitive is *terminal*, i.e. a leaf in the grammar and search tree, the enumerator scores the resulting program path for its validity over all transformations in the problem.

3 Experiment & Results

I produced a corpus of 32 problems with varying complexity, each of which contain 4 transformations $\{\delta_1, \delta_2, \delta_3, \delta_4\}$. 12 of these problems are annotated with ground-truth programs, used as training data. In general, there is a range and progression of complexity for the scenes within each problem (see figure 13).

For the experiment, we trained Θ against the set of annotated training problems. On each of the 20 un-annotated problems, we measured the number of transformations that the system could solve given the features of $\delta_1 \ldots \delta_n$ transformations. We ran each iteration on a modern quad-core laptop computer and limited the enumeration process to a duration of 3 minutes and a maximum b of 20.0. The hypothesis that could solve the highest number of transformations within this period would be recorded along with its cost and accuracy score (# of solved transformations).

Define:
$\mathcal{G}$: Grammar
τ: Desired Resultant Type
b: Current Cost Budget
β: Variable Context

function ENUMERATE($\mathcal{G}, \tau, b, \beta$)
  if $\tau = \lambda$ **then**
    $(\text{args}, \tau_{\text{new}}) \leftarrow \tau$
    $\beta \leftarrow \beta \cup \text{args}$
    $\lambda \leftarrow$ Nest n Abstractions for $n = |\text{args}|$
    return (Apply λ for all in ENUMERATE($\mathcal{G}, \tau_{\text{new}}, b, \beta$))
  else
    Candidate Primitives $\leftarrow \{(f, \tau_n, c) \in \mathcal{G} \mid (\tau_n = \tau) \wedge (c \leq b)\} \cup \beta$
    Programs $\leftarrow \{\}$
    for all P in Candidate Primitives **do**
      if $\tau_P = \lambda$ **then**
        Enumerated $\leftarrow$ ENUMERATE for all possible applications of τ_P
        Programs $\leftarrow$ Programs $\cup$ Enumerated
      else if τ_P is Terminal Primitive **then**
        Programs $\leftarrow$ Programs $\cup \{P\}$
      end if
    end for
    return Programs
  end if

Figure 11: A rough description of the enumeration algorithm.

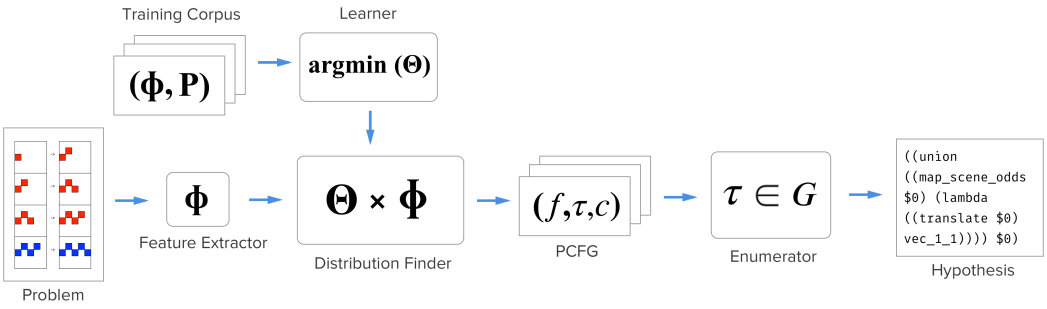

Figure 12: An illustration of the entire system.

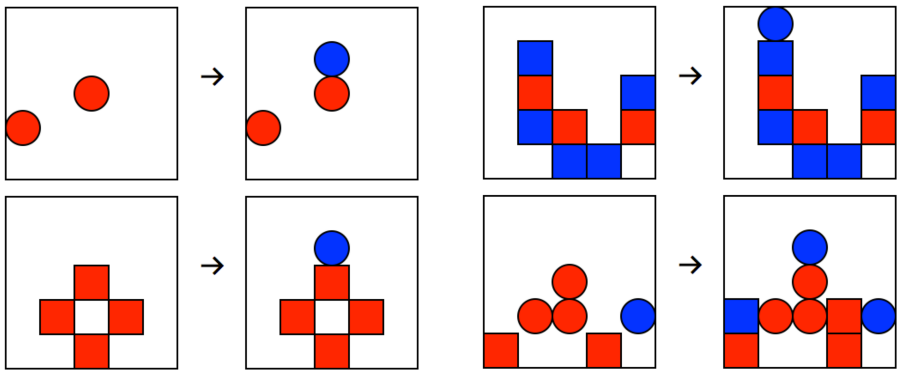

Figure 13: An example of a transformation complexity progression seen in the problem set. Notice the variance in scene features that could potentially produce "noise" for an otherwise trivial problem.

I compared the results of the trained model with that of an untrained baseline model, where the cost distribution of all $\{f \in \tau\} \in \mathcal{T}$ is uniform, or $\tilde{P}(f) = \exp(-|\mathcal{G}_\tau|)$. Because there are no features for this model, there is only one δ.

3.1 Results

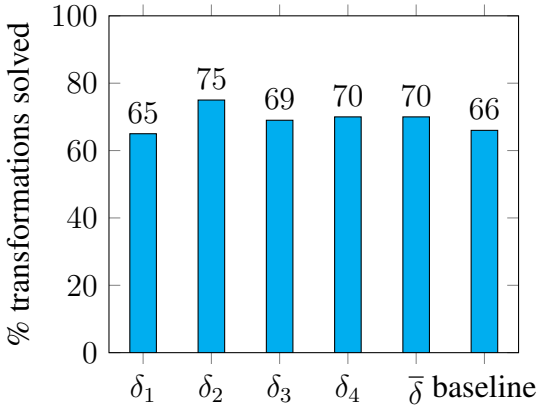

Figure 14: Accuracy of the different trials, in % transformations solved out of the $20 * 4$.

Both the trained and baseline systems are capable of finding completely valid solutions ($n = 4$ transformations) across all δ for 10 problems. Further, they can find partial solutions ($0 < n < 4$) for 8 problems, and no solutions for 2 problems ($n = 0$). On average, the trained system can

solve 70% of problems, and the baseline can solve 66% (figure 14).

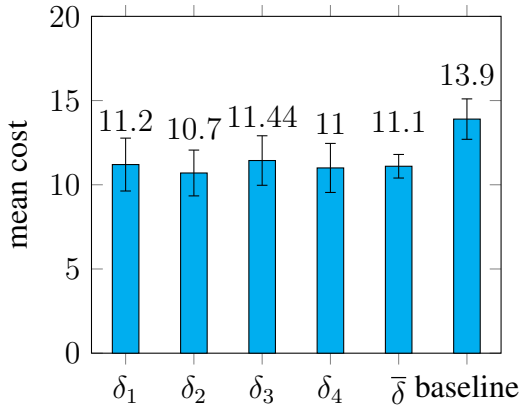

Figure 15: Mean cost for all trials $\in$ the set of complete solutions ($n = 10$).
NB: Lower cost is better, because $cost = -\log(P(f))$.

While there is not much difference in accuracy between the trained and baseline systems, there is a very notable difference in cost, and therefore, solving speed (figure 15). The mean cost is evaluated over all of the set of complete solutions ($n = 10$) so that only valid programs are compared for likelihood. The valid programs that the trained system finds are far more probable than the baseline, as the difference of cost between the baseline and $\overline{\delta} = 2.8$, therefore making the difference of probability on the order of $\exp(2.8) = 16.4$. Therefore, a valid program is approximately 16 times more likely to be enumerated in the trained system than the baseline, and thus will be found in approximately a factor of $1/16$ of the time.

4 Conclusion

4.1 Contribution

Our work provides a step forward in the development of two disciplines: Computational Cognitive Science and Human-Computer Interaction (HCI). From a Computational Cognitive Science perspective, we have proposed a system that can approximate human intuition in a relatively unexplored domain. The system provides a base design for solving a novel domain of problems

related to visual reasoning using little input criteria. We show that statistical inference and programming language theory can be successfully used for a domain previously only solvable by human intelligence. From a HCI perspective, we have demonstrated that a system using program synthesis techniques can be applied to domains of visual interaction such as image editing. Artificial Intelligence-guided interaction, especially in the creative and visual fields, is still in an early stage of development, and our work proposes a large portion of a practical system that could be used in the future.

4.2 Future Work

The most important step forward in this domain would be to enhance the dataset to solve upon. Building a large repertoire of standardized problems would provide a concrete benchmark that improved systems can compare against. Ideally, these problems would be derived from diverse real-world image editing tasks that have plenty of room for ambiguities and challenge.

Similarly, evaluating the dataset tasks with human subjects would prove a valuable benchmark to compare against. Using this basis of comparison can provide insight into the user's intentions, and how well the system can recover them.

There are also a number of enhancements that can be made both to the enumerator and feature extractor. Particularly, for more lengthy programs, the enumerator could be taught to learn to enumerate often-used program substructures along with the primitives in the grammar. As demonstrated in Liang et al., this approach accelerates the hypothesizing process and allows for more complex hypotheses. The model can then find nuanced relationships between substructures and higher-level features. For example, observing distinct "object clusters", such as a rectangle-shaped group composed of multiple squares, could have a strong correlation with substructures that can select and manipulate such clusters.

Finally, integrating the system into an image editor so that it can be used for real-world

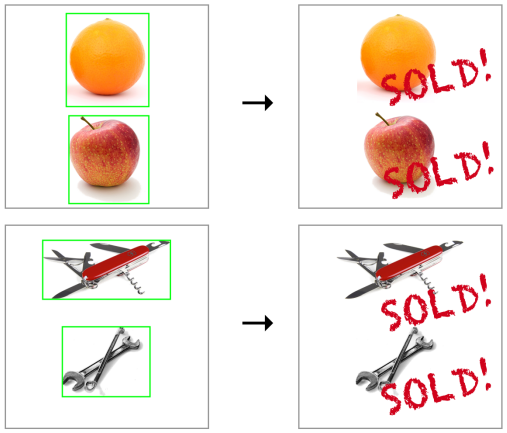

Figure 16: Objects in a scene could be batch-updated with additional overlays.

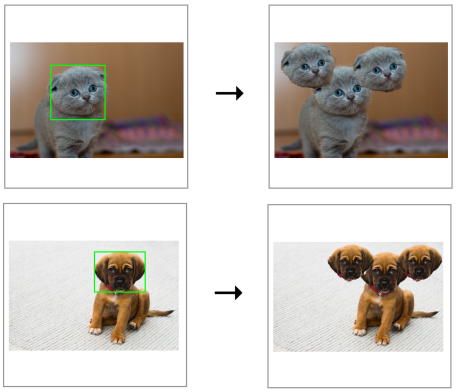

Figure 17: A computer vision model trained on face detection could invent a new species.

editing tasks would be a significant trial for its capabilities. The feature extractor could then be trained on a corpus of editor-native bounding boxes, dynamic painting instructions as discussed in Chugh et al., or even visual hierarchies inferred from an extractor using computer vision. Real-world editing tasks that would prove especially useful for the system could include appending label text to objects in an image (figure 16), or selecting and duplicating objects in a certain arrangement (figure 17).

References

1. Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DEEPCODER: LEARNING TO WRITE PROGRAMS. URL https://arxiv.org/pdf/1611.01989.pdf.

2. Ravi Chugh, Jacob Albers, and Mitchell Spradlin. Program Synthesis for Direct Manipulation Interfaces. URL http://people.cs.uchicago.edu/ rchugh/static/papers/sketch-n-sketch-draf

3. Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum. Sampling for bayesian program learning. In *Advances in Neural Information Processing Systems*, pages 1297–1305, 2016.

4. F. Fleuret, T. Li, C. Dubout, E. K. Wampler, S. Yantis, and D. Geman. Comparing machines and humans on a visual categorization test. *Proc. Natl. Acad. Sci. U.S.A.*, 108(43):17621–17625, Oct 2011.

5. Harry E Foundalis. Phaeaco: A cognitive architecture inspired by bongard's problems. 2006.

6. Sumit Gulwani. Automating String Processing in Spreadsheets Using Input-Output Examples. URL https://pdfs.semanticscholar.org/77b2/4ad6af8c68f7ba4f8386f612a27b7c902c

7. Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science (New York, N.Y.)*, 350 (6266):1332–8, dec 2015. ISSN 1095-9203. doi: 10.1126/science.aab3050. URL http://www.ncbi.nlm.nih.gov/pubmed/26659050.

8. Brenden M. Lake, Tomer D. Ullman, Joshua B. Tenenbaum, and Samuel J. Gershman. Building machines that learn and think like people. *CoRR*, abs/1604.00289, 2016. URL http://arxiv.org/abs/1604.00289.

9. Percy Liang, Michael I Jordan, and Dan Klein. Learning Programs: A Hierarchical Bayesian Approach. URL http://nlp.cs.berkeley.edu/pubs/Liang-Jordan-Klein_2010_Programs_paper.pd

10. Dianhuan Lin, Eyal Dechter, Kevin Ellis, Joshua Tenenbaum, and Stephen Muggleton. Bias reformulation for one-shot function induction. In *Proceedings of the Twenty-first European Conference on Artificial Intelligence*, pages 525–530. IOS Press, 2014.

11. D.J.C. MacKay. *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, 2003. ISBN 9780521642989. URL https://books.google.com/books?id=AKuMj4PN_EMC.

12. Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Tauman Kalai. A Machine Learning Framework for Programming by Example. URL http://proceedings.mlr.press/v28/menon13.pdf.

13. B.C. Pierce. *Types and Programming Languages*. Types and Programming Languages. MIT Press, 2002. ISBN 9780262162098. URL https://books.google.com/books?id=ti6zoAC9Ph8C.

14. United States Department of Labor. 27-1024 graphic designers, May 2016. URL https://www.bls.gov/oes/current/oes271024.htm.

1. Figures not cited are original by the Competition Entrant or in the public domain.