RobustFill: Neural Program Learning under Noisy I/O

Jacob Devlin [* 1] **Jonathan Uesato** [* 2] **Surya Bhupatiraju** [* 2] **Rishabh Singh** [1] **Abdel-rahman Mohamed** [1] **Pushmeet Kohli** [1]

Abstract

The problem of automatically generating a computer program from some specification has been studied since the early days of AI. Recently, two competing approaches for *automatic program learning* have received significant attention: (1) *neural program synthesis*, where a neural network is conditioned on input/output (I/O) examples and learns to generate a program, and (2) *neural program induction*, where a neural network generates new outputs directly using a *latent* program representation. Here, for the first time, we directly compare both approaches on a large-scale, real-world learning task and we additionally contrast to *rule-based program synthesis*, which uses hand-crafted semantics to guide the program generation. Our neural models use a modified attention RNN to allow encoding of variable-sized sets of I/O pairs, which achieve 92% accuracy on a real-world test set, compared to the 34% accuracy of the previous best neural synthesis approach. The synthesis model also outperforms a comparable induction model on this task, but we more importantly demonstrate that the strength of each approach is highly dependent on the evaluation metric and end-user application. Finally, we show that we can train our neural models to remain very robust to the type of noise expected in real-world data (e.g., typos), while a highly-engineered rule-based system fails entirely.

1. Introduction

The problem of *program learning*, i.e. generating a program consistent with some specification, is one of the oldest problems in machine learning and artificial intelligence

Input String	**Output String**
john Smith DOUG Q. Macklin Frank Lee (123) Laura Jane Jones	Smith, Jhn Macklin, Doug LEe, Frank Jones, Laura
Steve P. Green (9)	?
Program	
`GetToken(Alpha, -1) | ‘,’ | ‘ ’ | ToCase(Proper, GetToken(Alpha, 1))`	

Figure 1. An anonymized example from FlashFillTest with noise (typos). The goal of the task is to fill in the blank (i.e., '?' = 'Green, Steve'). Synthesis approaches achieve this by generating a program like the one shown. Induction approaches generate the new output string directly, conditioned on the the other examples.

Waldinger & Lee (1969); Manna & Waldinger (1975). The classical approach has been that of *rule-based program synthesis* (Manna & Waldinger, 1980), where a formal grammar is used to derive a program from a well-defined specification. Providing a formal specification is often more difficult than writing the program itself, so modern program synthesis methods generally rely on *input/output examples* (I/O examples) to act as an approximate specification. Modern rule-based synthesis methods are typically centered around hand-crafted function semantics and pruning rules to search for programs consistent with the I/O examples (Gulwani et al., 2012; Alur et al., 2013).

These hand-engineered systems are often difficult to extend and fragile to noise, so *statistical program learning* methods have recently gained popularity, with a particular focus on neural network models. This work has fallen into two overarching categories: (1) *neural program synthesis*, where the program is generated by a neural network conditioned on the I/O examples (Balog et al., 2016; Parisotto et al., 2017; Gaunt et al., 2016; Riedel et al., 2016), and (2) *neural program induction*, where network learns to generate the output directly using a *latent* program representation (Graves et al., 2014; 2016; Kurach et al., 2016; Kaiser & Sutskever, 2015; Joulin & Mikolov, 2015; Reed & de Freitas, 2016; Neelakantan et al., 2016). Although many of these papers have achieved impressive results on a variety of tasks, none have thoroughly compared induction and synthesis approaches on a real-world test set. In this work, we not only demonstrate strong empirical results compared

[*]Equal contribution [1]Microsoft Research, Redmond, Washington, USA [2]MIT, Cambridge, Massachusetts, USA. Correspondence to: Jacob Devlin <jdevlin@microsoft.com>.

Proceedings of the 34th International Conference on Machine Learning, Sydney, Australia, PMLR 70, 2017. Copyright 2017 by the author(s).

to past work, we also directly contrast the strengths and weaknesses of both *neural program learning* approaches for the first time.

The primary task evaluated for this work is a Programming By Example (PBE) system for string transformations similar to *FlashFill* (Gulwani et al., 2012; Gulwani, 2011). FlashFill allows Microsoft Excel end-users to perform regular expression-based string transformations using examples without having to write complex macros. For example, a user may want to extract zip codes from a text field containing addresses, or transform a timestamp to a different format. An example is shown in Figure 1. A user manually provides a small number of example output strings to convey the desired intent and the goal of FlashFill is to generalize the examples to automatically generate the corresponding outputs for the remaining input strings. Since the end goal is to emit the correct output strings, and not a program, the task itself is agnostic to whether a *synthesis* or *induction* approach is taken.

For modeling, we develop novel variants of the attentional RNN architecture (Bahdanau et al., 2014) to encode a variable-length unordered set of input-output examples. For program representation, we have developed a domain-specific language (DSL), similar to that of Gulwani et al. (2012), that defines an expressive class of regular expression-based string transformations. The neural network is then used to generate a program in the DSL (for synthesis) or an output string (for induction). Both systems are trained end-to-end using a large set of input-output examples and programs uniformly sampled from the DSL.

We compare our neural induction model, neural synthesis model, and the rule-based architecture of Gulwani et al. (2012) on a real-world FlashFill test set. We also inject varying amounts of noise (i.e., simulated typos) into the FlashFill test examples to model the robustness of different learning approaches. While the manual approaches work reasonably well for well-formed I/O examples, we show that its performance degrades dramatically in presence of even small amounts of noise. We show that our neural architectures are significantly more robust in presence of noise and moreover obtain an accuracy comparable to manual approaches even for non-noisy examples, thereby motivating the name *RobustFill*.

This paper makes the following key contributions:

- We present a novel variant of the attentional RNN architecture, which allows for encoding of a variable-size set of input-output examples.
- We evaluate the architecture on 205 real-world FlashFill instances and significantly outperform the previous best statistical system (92% vs. 34% accuracy).
- We compare the model to a hand-crafted synthesis algorithm and show that while both systems achieve similar performance on clean test data, our model is significantly more robust to realistic noise (with noise, 80% accuracy vs. 6% accuracy).
- We compare our neural synthesis architecture with a neural induction architecture, and demonstrate that each approach has its own strengths under different evaluation metrics and decoding constraints.

2. Related Work

There has been an abundance of recent work on neural program induction and synthesis.

Neural Program Induction: Neural Turing Machine (NTM) (Graves et al., 2014) uses a neural controller to read and write to an external memory tape using soft attention and is able to learn simple algorithmic tasks such as array copying and sorting. Stack-RNNs (Joulin & Mikolov, 2015) augment a neural controller with an external stack-structured memory and is able to learn algorithmic patterns of small description length. Neural GPU (Kaiser & Sutskever, 2015) presents a Turing-complete model similar to NTM, but with a parallel and shallow design similar to that of GPUs, and is able to learn complex algorithms such as long binary multiplication. Neural Programmer-Interpreters (Reed & de Freitas, 2016) teach a controller to learn algorithms from program traces as opposed to examples. Neural Random-Access Machines (Kurach et al., 2016) uses a continuous representation of 14 high-level modules consisting of simple arithmetic functions and reading/writing to a variable-size random-access memory to learn algorithmic tasks requiring pointer manipulation and dereferencing to memory. The domain of string transformations is different than the domains handled by these approaches and moreover, unlike RobustFill, these approaches need to be re-trained per problem instance.

Neural Program Synthesis: The most closely related work to ours uses a Recursive-Reverse-Recursive neural network (R3NN) to learn string transformation programs from examples (Parisotto et al., 2017), and is directly compared in Section 5.1. DeepCoder (Balog et al., 2016) trains a neural network to predict a distribution over possible functions useful for a given task from input-output examples, which is used to augment an external search algorithm. Unlike DeepCoder, RobustFill performs an end-to-end synthesis of programs from examples. Terpret (Gaunt et al., 2016) and Neural Forth (Riedel et al., 2016) allow programmers to write sketches of partial programs to express prior procedural knowledge, which are then completed by training neural networks on examples.

DSL-based synthesis: Non-statistical DSL-based synthesis approaches (Gulwani et al., 2012) exploit independence properties of DSL operators to develop a divide-and-conquer based search algorithm with several hand-crafted pruning and ranking heuristics (Polozov & Gulwani, 2015). In this work, we present a neural architecture to automatically learn an efficient synthesis algorithm. There is also some work on using learnt clues to guide the search in DSL expansions (Menon et al., 2013), but this requires hand-coded textual features of examples.

3. Problem Overview

We now formally define the problem setting and the domain-specific language of string transformations.

3.1. Problem Formulation

Given a set of input-output (I/O) string examples $(I_1, O_1), ..., (I_n, O_n)$, and a set of unpaired input strings $I^y_1, ..., I^y_m$, the goal of of this task is to generate the corresponding output strings, $O^y_1, ..., O^y_m$. For each example set, we assume there exists at least one program P that will correctly transform all of these examples, i.e., $P(I_1) \rightarrow O_1, ..., P(I^y_1) \rightarrow O^y_1, ...$ Throughout this work, we refer to (I_j, O_j) as *observed examples* and (I^y_j, O^y_j) as *assessment examples*. We use `InStr` and `OutStr` to generically refer to I/O examples that may be observed or assessed. We refer to this complete set of information as an *instance*:

$I_1 =$ `January`	$O_1 =$ `jan`
$I_2 =$ `February`	$O_2 =$ `feb`
$I_3 =$ `March`	$O_3 =$ `mar`
$I^y_1 =$ `April`	$O^y_1 =$ `apr`
$I^y_2 =$ `May`	$O^y_2 =$ `may`
$P =$ `ToCase(Lower, SubStr(1,3))`	

Intuitively, imagine that a (non-programmer) user has a large list of `InStr` which they wish to process in some way. The goal is to only require the user to manually create a small number of corresponding `OutStr`, and the system will generate the remaining `OutStr` automatically.

In the *program synthesis* approach, we train a neural model which takes $(I_1, O_1), ..., (I_n, O_n)$ as input and generates P as output, token-by-token. It is trained fully supervised on a large corpus of pairs of synthetic I/O examples and their respective programs. It is *not* conditioned on the assessment input strings I^y_j, but it could be in future work. At test time, the model is provided with new set of observed I/O examples and attempts to generate the corresponding P which it (maybe) has never seen in training. Crucially, the system can actually execute the generated P on each *observed* input string I_j and check if it produces O_j.[1] If not, it knows that P cannot be the correct program, and it can search for a different P. Of course, even if P is *consistent* on all observed examples, there is no guarantee that it will *generalize* to new examples (i.e., assessment examples). We can think of *consistency* as a necessary, but not sufficient, condition. The actual success metric is whether this program *generalizes* to the corresponding assessment examples, i.e., $P(I^y_j) = O^y_j$. There also may be multiple valid programs.

[1]This execution is deterministic, not neural.

In the *program induction* approach, we train a neural model which takes $(I_1, O_1), ..., (I_n, O_n)$ and I^y as input and generates O^y as output, character-by-character. Our current model decodes each assessment example independently. Crucially, the induction model makes no explicit use of program P at training or test time. Instead, we say that it induces a *latent* representation of the program. If we had a large corpus of real-world I/O examples, we could in fact train an induction model without any explicit program representation. Since such a corpus is not available, it is trained on the same synthesized I/O examples as the synthesis model. Note that since the program representation is latent, there is no way to measure consistency.

We can comparably evaluate both approaches by measuring *generalization accuracy*, which is the percent of test instances for which the system has successfully produced the correct `OutStr` for all assessment examples. For synthesis this means $P(I^y_j) = O^y_j \;\; \forall (I^y_j, O^y_j)$. For induction this means all O^y generated by the system are exactly correct. We typically use four observed examples and six assessment examples per test instance. All six must be exactly correct for the model to get credit.

3.2. The Domain Specific Language

The Domain Specific Language (DSL) used here to represent P models a rich set of string transformations based on substring extractions, string conversions, and constant strings. The DSL is similar to the DSL described in Parisotto et al. (2017), but is extended to include nested expressions, arbitrary constant strings, and a powerful regex-based substring extraction function. The syntax of the DSL is shown in Figure 2 and the formal semantics are presented in the supplementary material.

A program P : `string` $\Rightarrow$ `string` in the DSL takes as input a string and returns another string as output. The top-level operator in the DSL is the `Concat` operator that concatenates a finite list of string expressions e_i. A string expression e can either be a substring expression f, a nesting expression n, or a constant string expression. A substring expression can either be defined using two constant positions indices k_1 and k_2 (where negative indices denote positions from the right), or using the $\texttt{GetSpan}(\texttt{r}_1, \texttt{i}_\texttt{i}, \texttt{y}_1, \texttt{r}_2, \texttt{i}_2, \texttt{y}_2)$ construct that returns the substring between the $i_1^{\texttt{th}}$ occurrence of regex r_1 and the

$$\begin{array}{rcl}
\text{Program } p & := & \texttt{Concat}(\texttt{e}_1, \texttt{e}_2, \texttt{e}_3, ...) \\
\text{Expression } e & := & f \mid n \mid n_1(n_2) \mid n(f) \mid \texttt{ConstStr(c)} \\
\text{Substring } f & := & \texttt{SubStr}(\texttt{k}_1, \texttt{k}_2) \\
 & \mid & \texttt{GetSpan}(\texttt{r}_1, \texttt{i}_1, \texttt{y}_1, \texttt{r}_2, \texttt{i}_2, \texttt{y}_2) \\
\text{Nesting n} & := & \texttt{GetToken(t, i)} \mid \texttt{ToCase(s)} \\
 & \mid & \texttt{Replace}(\delta_1, \delta_2) \mid \texttt{Trim()} \\
 & \mid & \texttt{GetUpto(r)} \mid \texttt{GetFrom(r)} \\
 & \mid & \texttt{GetFirst(t, i)} \mid \texttt{GetAll(t)} \\
\text{Regex } r & := & t_1 \mid \cdots \mid t_n \mid \delta_1 \mid \cdots \mid \delta_m \\
\text{Type } t & := & \texttt{Number} \mid \texttt{Word} \mid \texttt{Alphanum} \\
 & \mid & \texttt{AllCaps} \mid \texttt{PropCase} \mid \texttt{Lower} \\
 & \mid & \texttt{Digit} \mid \texttt{Char} \\
\text{Case } s & := & \texttt{Proper} \mid \texttt{AllCaps} \mid \texttt{Lower} \\
\text{Position } k & := & -100, -99, ..., 1, 2, ..., 100 \\
\text{Index } i & := & -5, -4, -3, -2, 1, 2, 3, 4, 5 \\
\text{Character } c & := & \texttt{A} - \texttt{Z}, \texttt{a} - \texttt{z}, \texttt{0} - \texttt{9}, !?, @... \\
\text{Delimiter } \delta & := & \&, .?!@()[]\%\{\}/ :; \$\#"' \\
\text{Boundary } y & := & \texttt{Start} \mid \texttt{End}
\end{array}$$

Figure 2. Syntax of the string transformation DSL.

$i_2^{\texttt{th}}$ occurrence of regex r_2, where y_1 and y_2 denotes either the start or end of the corresponding regex matches. The nesting expressions allow for further nested string transformations on top of the substring expressions allowing to extract $k^{\texttt{th}}$ occurrence of certain regex, perform casing transformations, and replacing a delimiter with another delimiter. The notation $e_1 \mid e_2 \mid$ ... is sometimes used as a shorthand for $\texttt{Concat}(e_1, e_2, ...)$. The nesting and substring expressions take a string as input (implicitly as a lambda parameter). We sometimes refer expressions such as `ToCase(Lower)(v)` as `ToCase(Lower,v)`.

There are approximately 30 million unique string expressions e, which can be concatenated to create arbitrarily long programs. Any search method that does not encode inverse function semantics (either by hand or with a statistical model) cannot prune partial expressions. Thus, even efficient techniques like dynamic programming (DP) with black-box expression evaluation would still have to search over many millions of candidates.

3.3. Training Data and Test Sets

Since there are only a few hundred real-world FlashFill instances, the data used to train the neural networks was generated automatically. To do this, we use a strategy of random sampling and generation. First, we randomly sample programs from our DSL, up to a maximum length (10 expressions). Given a sampled program, we compute a simple set of heuristic requirements on the `InStr` such that the program can be executed without throwing an exception. For example, if an expression in the program retrieves the 4th number, the `InStr` must have at least 4 numbers. Then, each `InStr` is generated as a random sequence of ASCII characters, constrained to satisfy the requirements. The corresponding `OutStr` is generated by executing the program on the `InStr`.

For evaluating the trained models, we use *FlashFillTest*, a set of 205 real-world examples collected from Microsoft Excel spreadsheets, and provided to us by the authors of Gulwani et al. (2012) and Parisotto et al. (2017). Each FlashFillTest instance has ten I/O examples, of which the first four are used as observed examples and the remaining six are used as assessment examples.[2] Some examples of FlashFillTest instances are provided in the supplementary material. Intuitively, it is possible to generalize to a real-word test set using randomly synthesized training because the model is learning *function semantics*, rather than a particular data distribution.

4. Program Synthesis Model Architecture

We model program synthesis as a sequence-to-sequence generation task, along the lines of past work in machine translation (Bahdanau et al., 2014), image captioning (Xu et al., 2015), and program induction (Zaremba & Sutskever, 2014). In the most general description, we encode the observed I/O using a series of recurrent neural networks (RNN), and generate P using another RNN one token at a time. The key challenge here is that in typical sequence-to-sequence modeling, the input to the model is a single sequence. In this case, the input is a variable-length, unordered set of sequence pairs, where each pair (i.e., an I/O example) has an internal conditional dependency. We describe and evaluate several multi-attentional variants of the attentional RNN architecture (Bahdanau et al., 2014) to model this scenario.

4.1. Single-Example Representation

We first consider a model which only takes a single observed example (I, O) as input, and produces a program P as output. Note that this model is *not* conditioned on the assessment input I^y. In all models described here, P is generated using a sequential RNN, rather than a hierarchical RNN (Parisotto et al., 2017; Tai et al., 2015).[3] As demonstrated in Vinyals et al. (2015), sequential RNNs can be surprisingly strong at representing hierarchical structures.

We explore four increasingly complex model architectures, shown visually in Figure 3:

- **Basic Seq-to-Seq**: Each sequence is encoded with a non-attentional LSTM, and the final hidden state is

[2]In cases where less than 4 observed examples are used, only the 6 assessment examples are used to measure generalization.

[3]Even though the DSL does allow limited hierarchy, preliminary experiments indicated that using a hierarchical representation did not add enough value to justify the computational cost.

used as the initial hidden state of the next LSTM.

- **Attention-A**: O and P are attentional LSTMs, with O attending to I and P attending to O.[4]
- **Attention-B**: Same as Attention-A, but P uses a *double attention* architecture, attending to both O and I simultaneously.
- **Attention-C**: Same as Attention-B, but I and O are bidirectional LSTMs.

In all cases, the `InStr` and `OutStr` are processed at the character level, so the input to I and O are character embeddings. The vocabulary consists of all 95 printable ASCII tokens.

The inputs and targets for the P layer is the source-code-order linearization of the program. The vocabulary consists of 430 total program tokens, which includes all function names and parameter values, as well as special tokens for concatenation and end-of-sequence. Note that numerical parameters are also represented with embedding tokens. The model is trained to maximize the log-likelihood of the reference program P.

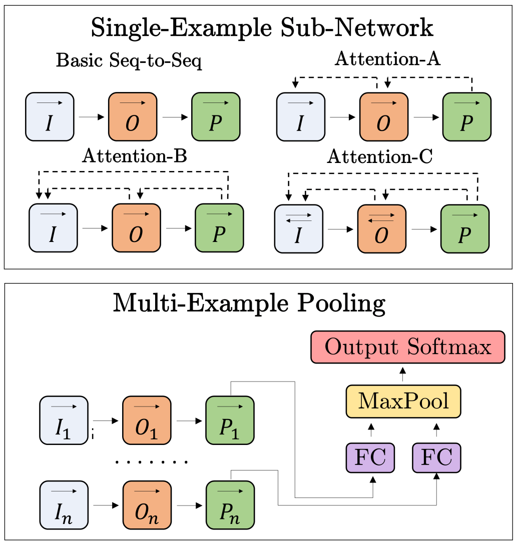

Figure 3. The network architectures used for program synthesis. A dotted line from x to y means that x attends to y.

4.2. Double Attention

Double attention is a straightforward extension to the standard attentional architecture, similar to the multimodal attention described in Huang et al. (2016). A typical attentional layer takes the following form:

$$\begin{aligned} c_i &= Attention(h_{i-1}, x_i, S) \\ h_i &= LSTM(h_{i-1}, x_i, c_i) \end{aligned}$$

Where S is the set of vectors being attended to, h_{i-1} is the previous recurrent state, and x_i is the current input. The $Attention()$ function takes the form of the "general" model from Luong et al. (2015). Double attention takes the form:

$$\begin{aligned} c_i^A &= Attention(h_{i-1}, x_i, S^A) \\ c_i^B &= Attention(h_{i-1}, x_i, s_i^A, S^B) \\ h_i &= LSTM(h_{i-1}, x_i, s_i^A, c_i^B) \end{aligned}$$

Note that s_i^A is concatenated to h_{i-1} when computing attention on S^B, so there is a directed dependence between the two attentions. Here, S^A is O and S^B is I. Exact details of the attentional formulas are given in the supplementary material.

4.3. Multi-Example Pooling

The previous section only describes an architecture for encoding a single I/O example. However, in general we assume the input to consist of multiple I/O examples. The number of I/O examples can be variable between test instances, and the examples are unordered, which suggests a pooling-based approach. Previous work (Parisotto et al., 2017) has pooled on the final encoder hidden states, but this approach cannot be used for attentional models.

Instead, we take an approach which we refer to as *late pooling*. Here, each I/O example has its own layers for I, O, and P (with shared weights across examples), but the hidden states of $P_1, ..., P_n$ are pooled at each timestep before being fed into a single output softmax layer. The architecture is shown at the bottom of Figure 3. We did not find it beneficial to add another fully-connected layer or recurrent layer after pooling.

Formally, the layers labeled "FC" and "MaxPool" perform the operation $m_i = \text{MaxPool}_{j \in n}(\tanh(W h_{ji}))$, where i is the current timestep, n is the number of observed examples, $h_{ji} \in \mathbb{R}^d$ is the output of P_j at the timestep i, and $W \in \mathbb{R}^{d \times d}$ is a set of learned weights. The layer denoted as "Output Softmax" performs the operation $y_i = \text{Softmax}(V m_i)$, where $V \in \mathbb{R}^{d \times v}$ is the output weight matrix, and v is the number of tokens in the program vocabulary. The model is trained to maximize the log-softmax of the reference program sequence, as is standard.

4.4. Hyperparameters and Training

In all experiments, the size of the recurrent and fully connected layers is 512, and the size of the embeddings is 128. Models were trained with plain SGD with gradient clipping. All models were trained for 2 million minibatch updates, where each minibatch contained 128 training instances (i.e., 128 programs with four I/O examples each). Each minibatch was re-sampled, so the model saw 256 mil-

[4] A variant where O and I are reversed performs significantly worse.

lion random programs and 1024 million random I/O examples during training. Training took approximately 24 hours on 2 Pascal Titan X GPUs, using an in-house toolkit. A small amount of hyperparameter tuning was done on a synthetic validation set that was generated like the training.

5. Program Synthesis Results

Once training is complete, the synthesis models can be decoded with a beam search decoder (Sutskever et al., 2014). Unlike a typical sequence generation task, where the model is decoded with a beam k and then only the 1-best output is taken, here all k-best candidates are executed one-by-one to determine consistency. If multiple program candidates are consistent with all observed examples, the program with the highest model score is taken as the output.[5] This program is referred to as P^*.

In addition to standard beam search, we also propose a variant referred to as "DP-Beam," which adds a search constraint similar to the dynamic programming algorithm mentioned in Section 3.3. Here, each time an expression is completed during the search, the partial program is executed in a black-box manner. If any resulting partial `OutStr` is not a string prefix of the observed `OutStr`, the partial program is removed from the beam. This technique is effective because our DSL is largely concatenative.

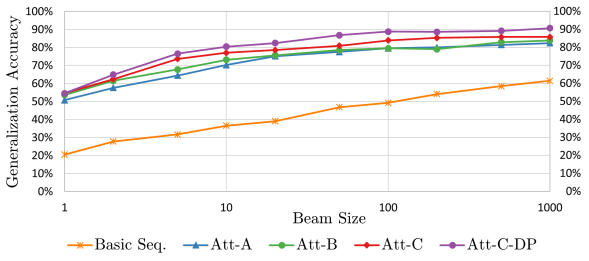

Figure 4. **Generalization Results**: Comparison of generalization accuracy for of several program synthesis architectures.

Generalization accuracy is computed by applying P^* to all six assessment examples. The percentage score reported in the figures represents the proportion of test instances for which a consistent program was found *and* it resulted in the exact correct output for all six assessment examples. Consistency is evaluated in Section 5.2.

Results are shown in Figure 4. The most evident result is that all attentional variants outperform the basic seq-to-seq model by a very large margin – roughly 25% absolute improvement. The difference between the three variants is smaller, but there is a clear improvement in accuracy as the models progress in complexity. Both Attention-B and Attention-C each add roughly 2-5% absolute accuracy, and this improvement appears even for a large beam. The DP-Beam variant also improves accuracy by roughly 5%. Overall, the best absolute accuracy achieved is 92% by Attention-C-DP w/ Beam=1000. Although we have not optimized our decoder for speed, the amortized end-to-end cost of decoding is roughly 0.3 seconds per test instance for Attention-C-DP w/ Beam=100 and four observed examples (89% accuracy), on a Pascal Titan X GPU.

5.1. Comparison to Past Work

Prior to this work, the strongest statistical model for solving FlashFillTest was Parisotto et al. (2017). The generalization accuracy is shown below:

System	**Beam**	
	100	**1000**
Parisotto et al. (2017)	23%	34%
Basic Seq-to-Seq	51%	56%
Attention-C	83%	86%
Attention-C-DP	89%	92%

We believe this improvement in accuracy is due to two key reasons. First, we used a seq-to-seq model with double attention instead of the tree-based R3NN model, which is difficult to train because of batching issues of tree-based structures for larger programs. Second, late pooling allows us to effectively incorporate powerful attention mechanisms into our model. Because the architecture in Parisotto et al. (2017) performed pooling at the I/O encoding level, it could not exploit the attention mechanisms which we show are critical to achieving high accuracy.

Comparison to the FlashFill implementation currently deployed in Microsoft Excel is given in Section 7.

5.2. Consistency vs. Generalization Results

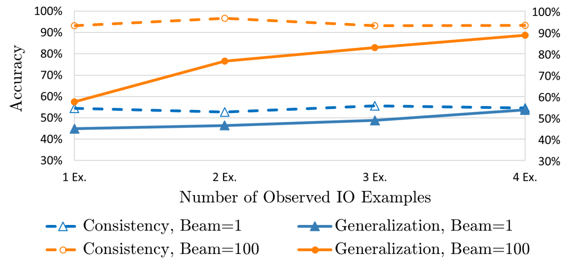

Figure 5. **Consistency vs. Generalization**: Results were obtained using Attention-C.

The conceptual difference between *consistency* and *generalization* is detailed in Section 3.1. Results for different beam sizes and different number of observed IO examples are presented in Figure 5. As expected, the generalization accuracy increases with the number of observed examples for both beam sizes, although this is significantly more pro-

[5] We tried several alternative heuristics, such as taking the shortest program, but these did not perform better.

nounced for a Beam=100. Interestingly, the consistency is relatively constant when the number of observed examples increases. There was no *a priori* expectation about whether consistency would increase or decrease, since more examples are consistent with fewer total programs, but also give the network a stronger input signal. Finally, we can see that the Beam=1 decoding only generates *consistent* output roughly 50% of the time, which implies that the latent function semantics learned by the model are still far from perfect.

6. Program Induction Results

An alternative approach to solving the FlashFill problem is *program induction*, where the output string is generated directly by the neural network without the need for a DSL. More concretely, we can train a neural network which takes as input a set of n observed examples $(I_1, O_1), ...(I_n, O_n)$ as well an unpaired `InStr`, I^y, and generates the corresponding `OutStr`, O^y. As an example, from Figure 1, I_1 = "`john Smith`", O_1 = "`Smith, Jhn`", I_2 = "`DOUG Q. Macklin`", ... , I^y = "`Steve P. Green`", O^y = "`Green, Steve`". Both approaches have the same end goal – determine the O^y corresponding to I^y – but have several important conceptual differences.

The first major difference is that the induction model does not use the program P anywhere. The synthesis model generates P, which is executed by the DSL to produced O^y. The induction model generates O^y directly by sequentially predicting each character. In fact, in cases where it is possible to obtain a very large amount of real-world I/O example sets, induction is a very appealing approach since it does not require an explicit DSL.[6] The core idea is the model learns some *latent* program representation which can generalize beyond a specific DSL. It also eliminates the need to hand-design the DSL, unless the DSL is needed to synthesize training data.

The second major difference is that program induction has no concept of *consistency*. As described previously, in program synthesis, a k-best list of program candidates is executed one-by-one, and the first program consistent with all observed examples is taken as the output. As shown in Section 5.2, if a consistent program can be found, it is likely to generalize to new inputs. Program induction, on the other hand, is essentially a standard sequence generation task akin to neural machine translation or image captioning – we directly decode O^y with a beam search and take the highest-scoring candidate as our output.

[6]In the results shown here, the induction model is trained on data synthesized with the DSL, but the model training is agnostic to this fact.

6.1. Comparison of Induction and Synthesis Models

Despite these differences, it is possible to model both approaches using nearly-identical network architectures. The induction model evaluated here is identical to synthesis Attention-A with late pooling, except for the following two modifications:

1. Instead of generating P, the system generates the new `OutStr` O^y character-by-character.
2. There is an additional LSTM to encode I^y. The decoder layer O^y uses double attention on O_j and I^y.

The induction network diagram is given in the supplementary material. Each (I^y, O^y) pair is decoded independently, but conditioned on all observed examples. The attention, pooling, hidden sizes, training details, and decoder are otherwise identical to synthesis. The induction model was trained on the same synthetic data as the synthesis models.

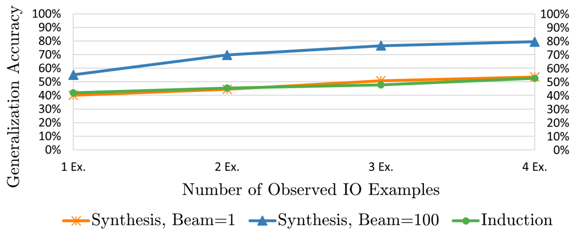

Figure 6. **Synthesis vs. Induction All-Example Accuracy**: The synthesis model uses Attention-A with standard beam search.

Results are shown in Figure 6. The induction model is compared to synthesis Attention-A using the same measure of generalization accuracy as previous sections – all six assessment examples must be exactly correct. Induction performs similarly to synthesis w/ beam=1, but both are significantly outperformed by synthesis w/ beam=100. The generalization accuracy achieved by the induction model is 53%, compared to 81% for the synthesis model. The induction model uses a beam of 3, and does not improve with a larger search because there is no way to evaluate candidates after decoding.

6.2. Average-Example Accuracy

All previous sections have used a strict definition of "generalization accuracy," requiring all six assessment examples to be exactly correct. We refer to this as *all-example* accuracy. However, another useful metric is to measure the total percent of correct assessment examples, averaged over all instances.[7] With this metric, generalizing on 5-out-of-6 assessment examples accumulates more credit than 0. We refer to this as *average-example* accuracy.

[7]The example still must be exactly correct – character edit rate is not measured here.

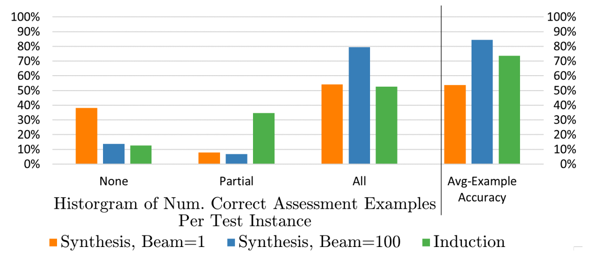

Figure 7. **Synthesis vs. Induction Average-Example Accuracy**

Average-example results are presented in Figure 7. The outcome matches our intuitions: Synthesis models tend to be "all or nothing," since it must find a *single* program that is jointly consistent with all observed examples. For both synthesis conditions, less than 10% of the test instances are partially correct. Induction models, on the other hand, have a much higher chance of getting *some* of the assessment examples correct, since they are decoded independently. Here, 33% of the test instances are partially correct. Examining the right side of the figure, the induction model shows relative strength under the average-example accuracy metric. However, in terms of absolute performance, the synthesis model still bests the induction model by 10%.

It is difficult to suggest which metric should be given more credence, since the utility depends on the downstream application. For example, if a user wanted to automatically fill in an entire column in a spreadsheet, they may prioritize all-example accuracy – *if* the system proposes a solution, they can be confident it will be correct for all rows. However, if the application instead offered auto-complete suggestions on a *per-cell* basis, then a model with higher average-example accuracy might be preferred.

7. Handling Noisy I/O Examples

For the FlashFill task, real-world I/O examples are typically manually composed by the user, so noise (e.g., typos) is expected and should be well-handled. An example is given in Figure 1.

Because neural network methods (1) are inherently probabilistic, and (2) operate in a continuous space representation, it is reasonable to believe that they can learn to be robust to this type of noise. In order to explicitly account for noise, we only made two small modifications. First, noise was synthetically injected into the training data using random character transformations.[8] Second, the best program P^* was selected by using *character edit rate* (CER) (Marzal & Vidal, 1993) to the observed examples, rather than exact match.[9]

Since the FlashFillTest set does not contain any noisy examples, noise was synthetically injected into the observed examples. All noise was applied with uniform random probability into the `InStr` or `OutStr` using character insertions, deletions, or substitutions. Noise is not applied to the *assessment* examples, as this would make evaluation impossible.

[8]This did not degrade the results on the noise-free test set.

[9]Standard beam is also used instead of DP-Beam.

We compare the models in this paper to the actual FlashFill implementation found in Microsoft Excel, as described in Gulwani et al. (2012). An overview of this model is described in Section 2. The results were obtained using a macro in Microsoft Excel 2016.

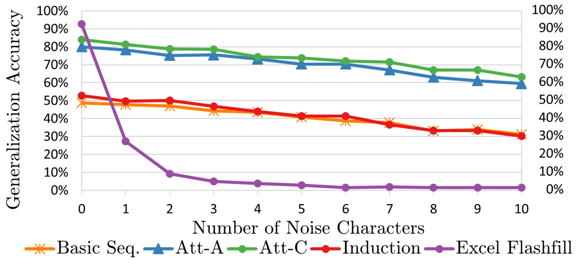

Figure 8. **Neural Synthesis vs. FlashFill**: All models use 4 I/O examples, and synthesis models use Beam=100

The noise results are shown in Figure 8. The neural models behave very similarly, each degrading approximately 2% absolute accuracy for each noise character introduced. The behavior of Excel FlashFill is quite different. Without noise, it achieves 92% accuracy,[10] matching the best result reported earlier in this paper. However, with just one or two characters of noise, Excel FlashFill is effectively "broken." This result is expected, since the efficiency of their algorithm is critically centered around exact string matching (Gulwani et al., 2012). We believe that this robustness to noise is one of the strongest attributes of DNN-based approaches to program synthesis.

8. Conclusions

We have presented a novel variant of an attentional RNN architecture for program synthesis which achieves 92% accuracy on a real-world Programming By Example task. This matches the performance of a hand-engineered system and outperforms the previous-best neural synthesis model by 58%. Moreover, we have demonstrated that our model remains robust to moderate levels of noise in the I/O examples, while the hand-engineered system fails for even small amounts of noise. Additionally, we carefully contrasted our neural program synthesis system with a neural program induction system, and showed that even though the synthesis system performs better on this task, both approaches have their own strengths under certain evaluation conditions. In particular, synthesis systems have an advantage when evaluating if *all* outputs are correct, while induction systems have strength when evaluating which system has the *most* correct outputs.

[10]FlashFill was manually developed on this exact set.

References

Alur, Rajeev, Bodik, Rastislav, Juniwal, Garvit, Martin, Milo MK, Raghothaman, Mukund, Seshia, Sanjit A, Singh, Rishabh, Solar-Lezama, Armando, Torlak, Emina, and Udupa, Abhishek. Syntax-guided synthesis. IEEE, 2013.

Bahdanau, Dzmitry, Cho, Kyunghyun, and Bengio, Yoshua. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Balog, Matej, Gaunt, Alexander L., Brockschmidt, Marc, Nowozin, Sebastian, and Tarlow, Daniel. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

Gaunt, Alexander L., Brockschmidt, Marc, Singh, Rishabh, Kushman, Nate, Kohli, Pushmeet, Taylor, Jonathan, and Tarlow, Daniel. Terpret: A probabilistic programming language for program induction. *CoRR*, abs/1608.04428, 2016.

Graves, Alex, Wayne, Greg, and Danihelka, Ivo. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Graves, Alex, Wayne, Greg, M, Reynolds, T, Harley, I, Danihelka, A, Grabska-Barwiska, SG, Colmenarejo, E, Grefenstette, T, Ramalho, J, Agapiou, and AP, Badia. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

Gulwani, Sumit. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*. ACM, 2011.

Gulwani, Sumit, Harris, William R, and Singh, Rishabh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 2012.

Huang, Po-Yao, Liu, Frederick, Shiang, Sz-Rung, Oh, Jean, and Dyer, Chris. Attention-based multimodal neural machine translation. In *Proceedings of the First Conference on Machine Translation, Berlin, Germany*, 2016.

Joulin, Armand and Mikolov, Tomas. Inferring algorithmic patterns with stack-augmented recurrent nets. In *NIPS*, pp. 190–198, 2015.

Kaiser, Lukasz and Sutskever, Ilya. Neural gpus learn algorithms. *CoRR*, abs/1511.08228, 2015.

Kurach, Karol, Andrychowicz, Marcin, and Sutskever, Ilya. Neural random-access machines. *ICLR*, 2016.

Luong, Minh-Thang, Pham, Hieu, and Manning, Christopher D. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

Manna, Zohar and Waldinger, Richard. Knowledge and reasoning in program synthesis. *Artificial intelligence*, 6 (2):175–208, 1975.

Manna, Zohar and Waldinger, Richard. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):90–121, 1980.

Marzal, Andres and Vidal, Enrique. Computation of normalized edit distance and applications. *IEEE transactions on pattern analysis and machine intelligence*, 1993.

Menon, Aditya Krishna, Tamuz, Omer, Gulwani, Sumit, Lampson, Butler W., and Kalai, Adam. A machine learning framework for programming by example. In *ICML*, pp. 187–195, 2013.

Neelakantan, Arvind, Le, Quov V., and Sutskever, Ilya. Neural programmer: Inducing latent programs with gradient descent. *ICLR*, 2016.

Parisotto, Emilio, Mohamed, Abdel-rahman, Singh, Rishabh, Li, Lihong, Zhou, Dengyong, and Kohli, Pushmeet. Neuro-symbolic program synthesis. *ICLR*, 2017.

Polozov, Oleksandr and Gulwani, Sumit. Flashmeta: a framework for inductive program synthesis. In *OOPSLA*, pp. 107–126, 2015.

Reed, Scott and de Freitas, Nando. Neural programmer-interpreters. *ICLR*, 2016.

Riedel, Sebastian, Bosnjak, Matko, and Rocktäschel, Tim. Programming with a differentiable forth interpreter. *CoRR*, abs/1605.06640, 2016.

Sutskever, Ilya, Vinyals, Oriol, and Le, Quoc V. Sequence to sequence learning with neural networks. In *NIPS*, 2014.

Tai, Kai Sheng, Socher, Richard, and Manning, Christopher D. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

Vinyals, Oriol, Kaiser, Łukasz, Koo, Terry, Petrov, Slav, Sutskever, Ilya, and Hinton, Geoffrey. Grammar as a foreign language. In *NIPS*, 2015.

Waldinger, Richard J. and Lee, Richard C. T. Prow: A step toward automatic program writing. In *IJCAI*, 1969.

Xu, Kelvin, Ba, Jimmy, Kiros, Ryan, Cho, Kyunghyun, Courville, Aaron C, Salakhutdinov, Ruslan, Zemel, Richard S, and Bengio, Yoshua. Show, attend and tell: Neural image caption generation with visual attention. In *ICML*, 2015.

Zaremba, Wojciech and Sutskever, Ilya. Learning to execute. *arXiv preprint arXiv:1410.4615*, 2014.