EXECUTION-GUIDED NEURAL PROGRAM SYNTHESIS

Anonymous authors
Paper under double-blind review

ABSTRACT

Neural program synthesis from input-output examples has attracted an increasing interest from both the machine learning and the programming language community. Most existing neural program synthesis approaches employ an encoder-decoder architecture, which uses an encoder to compute the embedding of the given input-output examples, as well as a decoder to generate the program from the embedding following a given syntax. Although such approaches achieve a reasonable performance on simple tasks such as FlashFill, on more complex tasks such as Karel, the state-of-the-art approach can only achieve an accuracy of around 77%. We observe that the main drawback of existing approaches is that the semantic information is greatly under-utilized. In this work, we propose two simple yet principled techniques to better leverage the semantic information, which are *execution-guided synthesis* and *synthesizer ensemble*. These techniques are general enough to be combined with any existing encoder-decoder-style neural program synthesizer. Applying our techniques to the Karel dataset, we can boost the accuracy from around 77% to more than 90%, reducing the error rate by around 60%.

1 INTRODUCTION

Program synthesis is a traditional challenging problem. Such a problem typically takes a *specification* as the input, and the goal is to generate a *program* within a target *domain-specific language* (DSL). One of the most interesting forms of the specifications is input-output examples, and there have been several applications, such as FlashFill (Gulwani, 2011; Gulwani et al., 2012).

Recently, there is an increasing interest of applying neural network approaches to tackle the program synthesis problem. For example, Devlin et al. have demonstrated that using an encoder-decoder-style neural network, their neural program synthesis algorithm called RobustFill can outperform the performance of the traditional non-neural program synthesis approach by a large margin on the FlashFill task (Devlin et al., 2017b).

Despite their promising performance, we identify several inefficiencies of such encoder-decoder-style neural program synthesis approaches. In particular, such a neural network considers program synthesis as a sequence generation problem; although some recent work take the syntactical information into consideration during program generation (Bunel et al., 2018; Rabinovich et al., 2017; Yin & Neubig, 2017; Parisotto et al., 2017; Xu et al., 2017), the semantic information, which is typically well-defined in the target DSL, is not effectively leveraged by existing work.

In light of this observation, in this work, we develop simple yet principled techniques that can be combined with any existing encoder-decoder-style neural program synthesizers. The main novel technique is called *execution-guided synthesis*. The basic idea is to view the program execution as a sequence of manipulations to transform each input state into the corresponding output state. In such a view, executing a *partial program* can result in intermediate states; thus, synthesizing the rest of the program can be conditioned on these intermediate states, so that the synthesizer can take the state changes into account in the followup program generation process. Therefore, we can leverage this idea to combine with any existing encoder-decoder-style neural synthesizer, and we observe that it can significantly improve the performance of the underlying synthesizers. In addition, we also propose a simple technique called *synthesizer ensemble*, which leverages the semantic information to ensemble multiple neural synthesizers, and we observe that this technique can further boost the performance.

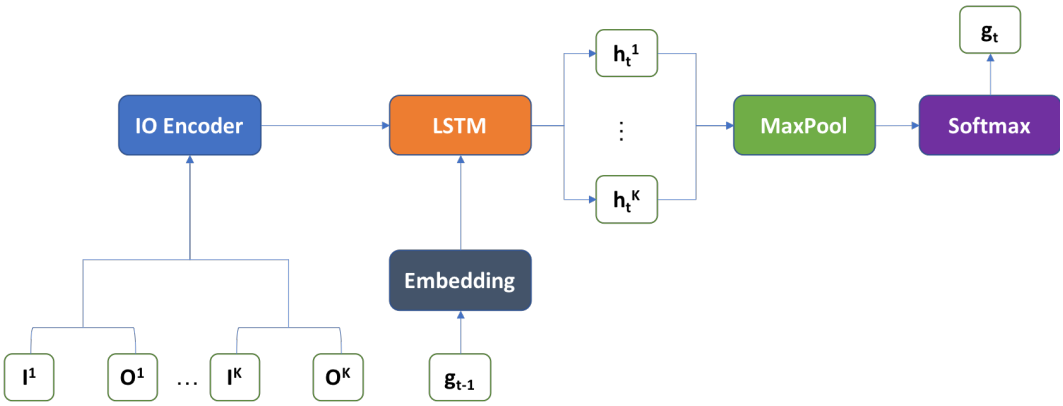

Figure 1: A general neural network architecture for input-output program synthesis.

We evaluate our techniques on the Karel task (Bunel et al., 2018; Devlin et al., 2017a), the largest publicly available benchmark for input-output program synthesis, on which the most performant model in the past can achieve only an accuracy of around 77% (Bunel et al., 2018). We observe that our proposed techniques can gain better performance than the previous state-of-the-art results. In particular, by combining both of our techniques, we can achieve an accuracy of more than 90%, which is over 13 percentage points better than the state-of-the-art results. In other words, we can reduce the error rate by around 60%. This shows that our approach is effective in boosting the performance of algorithms for the problem of neural program synthesis from input-output examples.

2 Neural Program Synthesis From Input-Ouput Examples

In this section, we first introduce the input-output program synthesis problem and existing encoder-decoder-style neural program synthesis approaches, then present an overview of our approaches.

2.1 Problem Definition

We follow the literature (Devlin et al., 2017b; Bunel et al., 2018; Chen et al., 2018) to formally define the input-output program synthesis problem below.

Problem Definition 1 (Program emulation) *Let $\mathcal{L}$ be the space of all valid programs in the domain-specific language (DSL). Given a set of input-output pairs $\{I^k, O^k\}_{k=1}^K$ (or $\{IO^K\}$ in short), where there exists a program $P \in \mathcal{L}$, such that $P(I^k) = O^k, \forall k \in \{1, ..., K\}$. Our goal is to compute the output O^{test} for a new test input I^{test}, so that $O^{\text{test}} = P(I^{\text{test}})$.*

Although the problem definition only requires to compute the output for a test input, a typical method is to synthesize a program $P' \in \mathcal{L}$ such that P' is consistent with all input-output pairs $\{IO^K\}$, and then use P' to compute the output. In this case, we say *program P' emulates the program P corresponding to $\{IO^K\}$*.

In particular, in this work, we are mainly interested in the following formulation of the problem.

Problem Definition 2 (Program synthesis) *Let $\mathcal{L}$ be the space of all valid programs in the domain-specific language (DSL). Given a training dataset of $\{IO^K\}_i$ for $i = 1, ..., N$, where N is the size of the training data, compute a* synthesizer Γ, *so that given a test input-output example set $\{IO^K\}_{\text{test}}$, the synthesizer $\Gamma(\{IO^K\}_{\text{test}}) = P$ produces a program P, which emulates the program corresponding to $\{IO^K\}_{\text{test}}$.*

2.2 Encoder-decoder-style Neural Program Synthesis Approaches

There have been many approaches proposed for different neural program synthesis tasks, and most of them follow an encoder-decoder-style neural network architecture (Bunel et al., 2018; Devlin et al.,

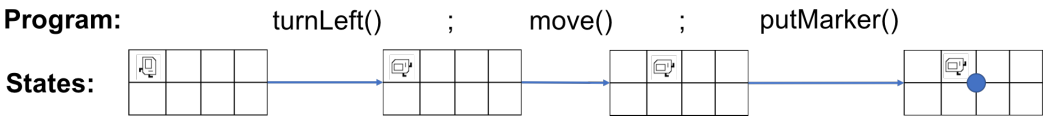

Figure 2: An example of the execution of partial programs to reach the target state in the Karel domain. The blue dot denotes the marker put by the Karel robot.

2017b; Parisotto et al., 2017). Figure 1 shows a general neural network architecture for input-output program synthesis. First, an encoder converts input-output examples $\{IO^K\}$ into an embedding. For example, RobustFill (Devlin et al., 2017b) deals with the string transformation tasks, thus it uses LSTMs as the encoder. For the Karel task, both inputs and outputs are 2D grids (see Figure 2); therefore, Bunel et al. (2018) employ a CNN as the encoder.

Once the IO embeddings are computed, these approaches employ an LSTM decoder to generate the programs conditioned on the embeddings. For program synthesis, one unique property is that the generated program should satisfy the synthax of $\mathcal{L}$. Therefore, a commonly used refinement is to filter syntactically invalid program prefixes during generation (Parisotto et al., 2017; Devlin et al., 2017b; Bunel et al., 2018).

In the above approaches, only the syntax information is leveraged; the semantics of $\mathcal{L}$ is not utilized. In particular, standard supervised training procedure could suffer from *program aliasing*: for the same input-output examples, there are multiple semantically equivalent programs, but all except the one provided in the training data will be penalized as wrong programs. To mitigate this issue, Bunel et al. (2018) propose to train with reinforcement learning, so that it rewards all semantically correct programs once they are fully generated. In our work, we demonstrate that we can leverage the semantic information in an effective way that provides a better performance.

2.3 An Overview of our Approaches

In this work, we propose two general and principled techniques that can improve the performance over existing work, which are *execution-guided synthesis* (Section 3) and *synthesizer ensemble* (Section 4). The main idea of our techniques is to better leverage the semantics of the language $\mathcal{L}$ during synthesis. Meanwhile, our techniques are compatible with any existing encoder-decoder-style neural program synthesis architecture. We will describe these techniques in detail in the following sections.

3 Execution-Guided Synthesis

Existing approaches generate the entire program only based on the input-output examples before execution. However, this is an inefficient use of the semantics of $\mathcal{L}$. For example, when a program consists of a sequence of statements, we can view the output to be a result by continuously executing each statement in the sequence to convert the input state into a sequence of intermediate states. Figure 2 illustrates such an example. From this perspective, instead of generating the whole program at once, we can generate one statement at a time based on the intermediate/output state pairs.

However, most interesting programs are not just sequential. In this work, we explore this idea using a general control-flow framework. In particular, given any language $\mathcal{L}$, we extend it with three classical types of control-flow: sequential, branching, and looping. The extended language is call $\mathcal{L}_{\text{ext}}$. Then, we develop our above idea based on $\mathcal{L}_{\text{ext}}$, called *execution-guided synthesis*. In the following, to make our discussion concise, we first formalize $\mathcal{L}_{\text{ext}}$ (Section 3.1), and then present the idea of execution-guided synthesis (Section 3.2).

3.1 The Formal Specification of the Extended Language $\mathcal{L}_{\text{ext}}$

In this work, we assume some additional control-flow syntax on top of $\mathcal{L}$. We define the extended language $\mathcal{L}_{\text{ext}}$ in Table 1. In particular, we assume that a code block B can be composed by a sequence of statements $\mathcal{S} \in \mathcal{L}$ or sub

$$
\begin{array}{rcl}
P, B & := & \bot \mid \mathcal{S} \mid B; B \\
 & \mid & \textbf{if } \mathcal{C} \textbf{ then } B \textbf{ else } B \textbf{ fi} \\
 & \mid & \textbf{while } \mathcal{C} \textbf{ do } B \textbf{ end} \\
\mathcal{S}, \mathcal{C} & \in & \mathcal{L}
\end{array}
$$

Table 1: Syntax of $\mathcal{L}_{\text{ext}}$.

$$\text{S-S}\ \frac{\langle \mathcal{S}, s\rangle \Downarrow s'}{\langle \mathcal{S}, s\rangle \rightarrow \langle \bot, s'\rangle} \qquad \text{S-Seq}\ \frac{\langle B_1, s\rangle \rightarrow \langle B_1', s'\rangle}{\langle B_1; B_2, s\rangle \rightarrow \langle B_1'; B_2, s'\rangle}$$

$$\text{S-Seq-Bot}\ \langle \bot; B_2, s\rangle \rightarrow \langle B_2, s\rangle \qquad \text{S-If}\ \frac{\langle \mathcal{C}, s\rangle \Downarrow b \qquad i = \begin{cases} 1 & b \text{ is true} \\ 2 & b \text{ is false} \end{cases}}{\langle \textbf{if } \mathcal{C} \textbf{ then } B_1 \textbf{ else } B_2 \textbf{ fi}, s\rangle \rightarrow \langle B_i, s\rangle}$$

$$\text{S-While}\ \langle \textbf{while } \mathcal{C} \textbf{ do } B \textbf{ end}, s\rangle \rightarrow \langle \textbf{if } C \textbf{ then } B; \textbf{while } C \textbf{ do } B \textbf{ end else } \bot \textbf{ fi}, s\rangle$$

Table 2: Semantics rules $\langle B, s\rangle \rightarrow \langle B', s'\rangle$ for $\mathcal{L}_{\text{ext}}$.

Algorithm 1 Execution-guided synthesis (sequential case)

1: $s_{\mathbf{i}}^k \leftarrow I^k$ for $k = 1, ..., K$
2: $s_{\mathbf{o}}^k \leftarrow O^k$ for $k = 1, ..., K$
3: $P \leftarrow \bot$
4: $\mathcal{S} \leftarrow \Gamma(\{(s_{\mathbf{i}}^k, s_{\mathbf{o}}^k)\}_{k=1}^K)$
5: **while** $S \neq \bot$ **do**
6: $\quad \mathcal{S} \leftarrow \Gamma(\{(s_{\mathbf{i}}^k, s_{\mathbf{o}}^k)\}_{k=1}^K)$
7: $\quad P \leftarrow P; \mathcal{S}$
8: $\quad \langle \mathcal{S}, s_{\mathbf{i}}^k\rangle \rightarrow \langle \bot, s_{\mathbf{new}}^k\rangle$ for $k = 1, ..., K$
9: $\quad s_{\mathbf{i}}^k \leftarrow s_{\mathbf{new}}^k$ for $k = 1, ..., K$
10: **end while**
11: **return** P

code blocks, and each code block can also be an if-statement or a while-statement. We use $\mathcal{C}$ to indicate a condition expression in $\mathcal{L}$, and $\bot$ to indicate the termination of a program execution.

The semantics of $\mathcal{L}_{\text{ext}}$ is specified in Table 2. These rules are largely standard. The judgment $\langle B, s\rangle \rightarrow \langle B', s'\rangle$ indicates a small-step execution of program B over state s to result in a new program B' and a new state s'. The judgments $\langle \mathcal{S}, s\rangle \Downarrow s'$ and $\langle C, s\rangle \Downarrow b$ capture the big-step execution in $\mathcal{L}$ that statement $\mathcal{S}$ evaluates to a new state s' from s, and condition $\mathcal{C}$ evaluates to a boolean value b from s. Following the semantics of $\mathcal{L}_{\text{ext}}$, we can formally define a program execution.

Definition 1 (Program execution) *Given a program $P \in \mathcal{L}_{\text{ext}}$ and an input I, the execution is a sequence $s_0 ... s_T$, such that (1) $s_0 = I$; (2) $B_0 = P$; (3) $\langle B_i, s_i\rangle \rightarrow \langle B_{i+1}, s_{i+1}\rangle$ for $i = 0, ..., T-1$; and (4) $B_T = \bot$. The output of the program is $O = s_T$.*

3.2 Execution-Guided Synthesis Algorithm

In definition 1, we can observe that the initial and final states are simply two special states provided as the input-output examples of the synthesis problem. Thus, a synthesizer Γ for input-output pairs should also be able to take any state-pairs as inputs. Our execution-guided synthesis algorithm takes advantage of this fact to improve the performance of the synthesizer. In the following, we discuss three cases from the easiest to the hardest to present our approach.

Sequential programs. We now consider the simplest case, where the program is in the form of $\mathcal{S}_1; ...; \mathcal{S}_T$, to illustrate the basic idea of execution-guided synthesis algorithm. We present the algorithm in Algorithm 1. Assuming the input-output examples are $\{IO^K\}$, we treat them as K state-pairs $\{(s_{\mathbf{i}}^k, s_{\mathbf{o}}^k)\}_{k=1}^K$, where $s_{\mathbf{i}}^k = I^k, s_{\mathbf{o}}^k = O^k$ (line 1-2). The synthesized program is initially empty (line 3). Then the algorithm iteratively generates one statement $\mathcal{S}$ at a time (line 6), and appends it to the end of P (line 7). Importantly, the algorithm executes the newly generated statement $\mathcal{S}$ to transit $s_{\mathbf{i}}^k$ into $s_{\mathbf{new}}^k$ (line 8-9). Therefore, in the subsequent iteration, the synthesizer can start from the new states $s_{\mathbf{new}}^k$ after executing the partial program P generated so far. In doing so, the synthesizer can see all intermediate states to better adjust the followup synthesis strategies to improve the overall synthesis performance.

Algorithm 2 Execution-guided synthesis (mainbody to deal with branching)

Require: State-pair set $\mathcal{I}$

1: ...
2: **if** $\Gamma(\mathcal{I})$ predicts **if**-token **then**
3: $\quad \mathcal{C} \leftarrow \Gamma(\mathcal{I})$
4: $\quad \mathcal{I}_{\mathrm{t}} \leftarrow \{(s_{\mathbf{i}}, s_{\mathbf{o}}) \in \mathcal{I} | \langle \mathcal{C}, s_{\mathbf{i}} \rangle \Downarrow \mathrm{true}\}$
5: $\quad \mathcal{I}_{\mathrm{f}} \leftarrow \{(s_{\mathbf{i}}, s_{\mathbf{o}}) \in \mathcal{I} | \langle \mathcal{C}, s_{\mathbf{i}} \rangle \Downarrow \mathrm{false}\}$
6: $\quad$ run execution-guided synthesis over $\mathcal{I}_{\mathrm{t}}$ to generate B_{t} until Γ predicts **else**-token
7: $\quad$ run execution-guided synthesis over $\mathcal{I}_{\mathrm{f}}$ to generate B_{f} until Γ predicts **fi**-token
8: $\quad \mathcal{I}'_{\mathrm{t}} \leftarrow \{(s_{\mathbf{new}}, s_{\mathbf{o}}) | (s_{\mathbf{i}}, s_{\mathbf{o}}) \in \mathcal{I}_{\mathrm{t}} \wedge \langle B_{\mathrm{t}}, s_{\mathbf{i}} \rangle \Downarrow s_{\mathbf{new}}\}$
9: $\quad \mathcal{I}'_{\mathrm{f}} \leftarrow \{(s_{\mathbf{new}}, s_{\mathbf{o}}) | (s_{\mathbf{i}}, s_{\mathbf{o}}) \in \mathcal{I}_{\mathrm{f}} \wedge \langle B_{\mathrm{f}}, s_{\mathbf{i}} \rangle \Downarrow s_{\mathbf{new}}\}$
10: $\quad \mathcal{I} \leftarrow \mathcal{I}'_{\mathrm{t}} \cup \mathcal{I}'_{\mathrm{f}}$
11: $\quad P \leftarrow P; \mathbf{if}\ C\ \mathbf{then}\ B_{\mathrm{t}}\ \mathbf{else}\ B_{\mathrm{f}}\ \mathbf{fi}$
12: **end if**
13: ...

Branching programs. Dealing with if-statements is slightly more complicated than sequential programs, since in an if-statement, not all statements will be executed on all inputs. Following Algorithm 1 naively, we have to use Γ to synthesize the entire if-statement before being able to execute the partially generated program to derive intermediate states.

Therefore, in Algorithm 2, we extend the above idea to handle if-statements. When the next predicted token is an **if**-token (line 2), our execution-guided synthesizer first predicts the condition of the if-statement $\mathcal{C}$ (line 3). Then, we evaluate $\mathcal{C}$ over all state-pairs. Based on the evaluation results, we can divide the IO pairs into two sets $\mathcal{I}_{\mathrm{t}}$ and $\mathcal{I}_{\mathrm{f}}$ (line 4-5), so that all states in the former meet the branching condition to go to the true branch, and all states in the latter go to the false branch. Therefore, in the followup synthesis, we do not need to consider $\mathcal{I}_{\mathrm{f}}$ (or $\mathcal{I}_{\mathrm{t}}$) when synthesizing the true branch (or the false branch) (line 6-7). Note that in line 6-7, synthesizing both true-branch and false-branch can employ execution-guided synthesis algorithm to leverage intermediate states. Once we have done the synthesis of both branches, we can execute the generated branches to get the new states $\mathcal{I}$ (line 8-10), and append the newly generated if-statement to the end of the program.

In Algorithm 2, we use $\langle B, s \rangle \Downarrow s'$ to indicate a big-step execution of code block B over state s to get s'. In particular, this means that $\langle B, s \rangle \rightarrow \langle B_1, s_1 \rangle \rightarrow ... \rightarrow \langle \perp, s' \rangle$.

Looping programs. The remaining problem is to handle while-statements. Due to the rule S-While (see Table 2), a while-statement

$$\mathbf{while}\ \mathcal{C}\ \mathbf{do}\ B\ \mathbf{end} \tag{1}$$

is equivalent to

$$\mathbf{if}\ \mathcal{C}\ \mathbf{then}\ (B; \mathbf{while}\ \mathcal{C}\ \mathbf{do}\ B\ \mathbf{end})\ \mathbf{else}\ \perp\ \mathbf{fi} \tag{2}$$

Therefore, we can employ a procedure similar to Algorithm 2 once a **while**-token is predicted. However, there are two differences. First, in (2), the false-branch is empty, thus we do not need to deal with the false-branch. Second, although the true-branch is $B; \mathbf{while}\ \mathcal{C}\ \mathbf{do}\ B\ \mathbf{end}$, once we have generated B, we do not need to generate the rest of the true-branch, since both $\mathcal{C}$ and B have been generated.

Remarks. Note that our execution-guided synthesis algorithm can be applied to any neural synthesizer Γ, and we can train the synthesizer Γ using any supervised or reinforcement learning algorithms that have been proposed before (Devlin et al., 2017b; Bunel et al., 2018). In our evaluation, we demonstrate that this technique helps boost the performance of different existing training algorithms.

4 Synthesizer Ensemble

In our experiments, we observe that when we use different random initializations of the synthesizer during training, even if the synthesizer architectures are the same, they will be effective on different subsets of the dataset, although the overall prediction accuracy is similar to each other. Thus, a natural idea is to train multiple synthesizers, and ensemble them to build a more performant synthesizer.

Different from other deep learning tasks, for program synthesis task, without knowing the ground truth, we can already filter out those wrong predictions that cannot satisfy the input-output specification. Thus, we ensemble multiple synthesizers as follows: we run all synthesizers to obtain multiple programs, and select from programs that are consistent with all input-output examples. This provides us with a better chance to select the correct final prediction that generalizes to held-out IO pairs.

The main subtlety of such an approach is to deal with the case when multiple generated programs all satisfy the input-output examples. In this work, we consider several alternatives as follows:

- **Majority vote.** We can choose the most frequently predicted program as the final prediction.
- **Shortest.** Following the Occam's razor principle, we can choose the shortest program as the final prediction.

5 EVALUATION

In this section, we demonstrate the effectiveness of our approaches on the Karel dataset (Pattis, 1981; Bunel et al., 2018). We first introduce the task, discuss the experimental details, then present the results.

5.1 THE KAREL TASK

Karel is an educational programming language proposed in the 1980s (Pattis, 1981). Using this language, we can control a robot to move inside a 2D grid world and modify the world state, and our goal is to synthesize a program given a small number of input and output grids as the specification. Such tasks have been used in Stanford CS introductory courses (CS106A, 2018) and the Hour of Code (HoC, 2018), and have been studied recently in several neural program synthesis works (Devlin et al., 2017a; Bunel et al., 2018; Shin et al., 2018). Figure 2 shows an example in the Karel domain. We provide the grammar specification and the state representation in Appendix A. In particular, the Karel DSL includes control flow constructs such as conditionals and loops, which is more challenging than problems well-studied before, such as FlashFill (Gulwani, 2011; Devlin et al., 2017b).

Our evaluation follows the setup in (Bunel et al., 2018). We train and evaluate our approaches on their dataset, which is built by randomly sampling programs from the DSL. For each program, 5 IO pairs serve as the specification, and the sixth one is the held-out test sample. In total, there are 1,116,854 programs for training, 2,500 in the validation set, and 2,500 in the test set. We evaluate the following two metrics, which are the same as in (Bunel et al., 2018):

- **Exact Match.** The predicted program is an *exact match* if it is the same as the ground truth.
- **Generalization.** The predicted program is a *generalization* if it satisfies the input-output examples in both the specification and the held-out examples.

5.2 MODEL DETAILS

We employ the same neural network architecture as in (Bunel et al., 2018) to synthesize the programs, which is briefly discussed in Section 2.2. During the inference time, we set the beam size $B = 64$, and select the one with the highest prediction probability from the remaining programs. More details can be found in Appendix B.

5.3 RESULTS

We present our main results in Table 3. For reference, we include MLE and RL approaches in (Bunel et al., 2018), which were the state-of-the-art results on the Karel task for the generalization and exact-match metrics respectively. We first apply our ensemble approaches to these approaches, and observe that the performance is significantly boosted by up to 7 percentage points.

We next observe that our execution-guided synthesis alone can significantly improve the generalization accuracy over all approaches from (Bunel et al., 2018), even after we accompany their approaches with our ensemble techniques. In particular, without the ensemble, "Exec" already improves "RL" by

	Generalization	Exact Match
MLE (Bunel et al., 2018)	71.91%	39.94%
RL (Bunel et al., 2018)	77.12%	32.17%
MLE + Ensemble (S)	78.80%	45.56%
MLE + Ensemble (MV)	78.80%	45.84%
RL + Ensemble (S)	84.84%	45.80%
RL + Ensemble (MV)	84.00%	45.80%
Exec	85.08%	40.88%
Exec + RL	86.04%	39.40%
Exec + Ensemble (S)	90.60%	**46.36%**
Exec + Ensemble (MV)	**90.68%**	44.60%

Table 3: Accuracy on the Karel test set. "MLE" and "RL" indicates the two approaches proposed in (Bunel et al., 2018); "Exec" to indicate execution-guided synthesis, and "Ensemble" for synthesizer ensemble. For the ensemble, we report the 10-model ensemble results. "Exec" indicates exeuction-guided synthesis; "S" indicates the shortest principle; and "MV" indicates the majority voting principle.

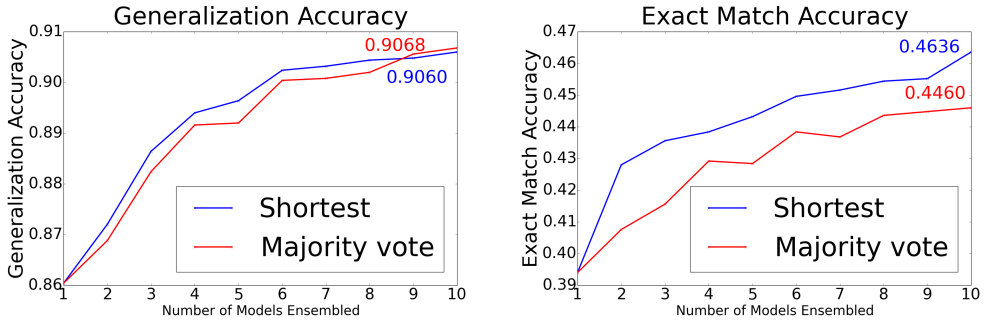

Figure 3: Results of the ensemble model trained with our Exec approach. Left: generalization accuracy. Right: exact match accuracy.

8 points on generalization accuracy; and when ensemble approaches are applied to "RL", this gap is shrunk, but still positive.

Similar to (Bunel et al., 2018), we can also further train our Exec model using the RL technique, which improves the generalization accuracy by 1 point, while slightly decreases the exact match accuracy. These results show that utilizing intermediate execution states alone is already an effective approach to boost the performance.

Finally, we apply our ensemble approaches on top of Exec. We observe that this can further improve the generalization accuracy by 4.6 points, and the exact match accuracy by 5.5 points from the best single model. These results show that our ensemble approaches consistently boosts the performance, regardless of the underlying models used for ensembling.

In addition, we investigate the performance of different number of models in the ensemble, and present the results in Figure 3. We observe that when there are fewer models, i.e., less than 9, using the shortest principle is more effective than using the majority vote principle. However, when there are more models, majority vote starts to achieve a slightly better generalization accuracy than the shortest principle. This is reasonable, since when there are too few models, there might not be enough effective models to form the majority.

Interestingly, we observe that applying RL to Exec+Ensemble does not improve the performance over Exec+Ensemble; the accuracy difference is within ± 0.5 point using different metrics. This may be due to that the improvement from ensemble hides the improvement from RL. More details and results about the ensemble can be found in Appendix C.

To summarize, we make the following key observations:

1. Our execution-guided synthesis technique can effectively improve previous approaches, which only use the syntactic information, or the final program execution outputs.
2. Our ensemble approaches can effectively improve the performance regardless of the underlying models being used.
3. The different modules of our proposed approaches can work independently to improve the performance, and thus they can be applied independently to other tasks as well.
4. By combining all our novel techniques, we improve the state-of-the-art on the Karel task by 13.56 points (generalization) and 6.42 points (exact match). In particular, we reduce error rates by 59.2% (generalization) and 10.7% (exact match) respectively.

6 Related Work

Synthesizing a program from input-output examples is an important challenging problem with many applications (Devlin et al., 2017b; Gulwani et al., 2012; Gulwani, 2011; Bunel et al., 2018; Chen et al., 2018; Cai et al., 2017; Li et al., 2017; Reed & De Freitas, 2016; Zaremba et al., 2016; Zaremba & Sutskever, 2015; Fox et al., 2018; Xiao et al., 2018; Ganin et al., 2018). There has been an emerging interest in studying neural networks for program synthesis. A line of work studies training a neural network to directly generate the outputs given the inputs (Devlin et al., 2017b;a; Graves et al., 2014; Joulin & Mikolov, 2015; Kaiser & Sutskever, 2015). In particular, Devlin et al. study the Karel domain (Devlin et al., 2017a). However, as shown in (Devlin et al., 2017a), this approach is incapable of handling the case when the number of input-output examples is small, and is hard to generalize to unseen inputs.

Recent work study using neural networks to generate programs in a domain-specific language (DSL) from a few input-output examples (Devlin et al., 2017b; Bunel et al., 2018; Parisotto et al., 2017; Polosukhin & Skidanov, 2018). Several work synthesize programs for FlashFill tasks, which are in the string transformation domain (Devlin et al., 2017b; Parisotto et al., 2017). Other work synthesize programs in a LISP-style DSL for array manipulation (Polosukhin & Skidanov, 2018). However, these DSLs only include sequential programs, and do not support more complex control flows such as loops and conditionals in our studied Karel problem. Prior works also consider incorporating syntax constraints and information from program execution to facilitate program synthesis (Devlin et al., 2017b; Wang et al., 2018; Bunel et al., 2018). However, all these works generate the whole program, and use its execution results to guide the synthesis process; in contrast, our work leverages more fine-grained yet generic semantic information that can be gathered during executing programs in most imperative languages. As a result, our approach's performance is significantly better than previous work (Bunel et al., 2018).

In contrast to training a neural network to generate the entire program, a recent line of research studies using a neural network to guide the symbolic program search based on the input-output specification, so that the search process prioritizes the operators that have higher domain-specific scores predicted by the neural networks (Balog et al., 2017; Vijayakumar et al., 2018). Instead of predicting such domain-specific scores to guide the program search, we directly incorporate the domain knowledge by executing partial programs, and utilize the execution results for program generation of the neural network synthesizer.

7 Conclusion

In this work, we propose two general and principled techniques to better leverage the semantic information for neural program synthesis: (1) execution-guided synthesis; and (2) synthesizer ensemble. On a rich DSL with complex control flows, we achieve a significant performance gain over the existing work, which demonstrates that utilizing the semantic information is crucial in boosting the performance of neural program synthesis approaches. We believe that our techniques are also beneficial to other program generation applications, and we consider extending our techniques to handle programming languages with richer semantics as important future work. At the same time, we have observed that utilizing existing reinforcement learning techniques does not provide much performance gain when combined with our approaches. We believe that there is plenty of room left for further improvement, and we are also interested in exploring this problem in the future.

REFERENCES

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *ICLR*, 2017.

Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. In *International Conference on Learning Representations*, 2018.

Jonathon Cai, Richard Shin, and Dawn Song. Making neural programming architectures generalize via recursion. In *ICLR*, 2017.

Xinyun Chen, Chang Liu, and Dawn Song. Towards synthesizing complex programs from input-output examples. In *ICLR*, 2018.

CS106A. Stanford cs106a course page. https://see.stanford.edu/Course/CS106A, 2018.

Jacob Devlin, Rudy R Bunel, Rishabh Singh, Matthew Hausknecht, and Pushmeet Kohli. Neural program meta-induction. In *Advances in Neural Information Processing Systems*, pp. 2077–2085, 2017a.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy I/O. In *ICML*, 2017b.

Roy Fox, Richard Shin, Sanjay Krishnan, Ken Goldberg, Dawn Song, and Ion Stoica. Parametrized hierarchical procedures for neural programming. In *ICLR*, 2018.

Yaroslav Ganin, Tejas Kulkarni, Igor Babuschkin, SM Eslami, and Oriol Vinyals. Synthesizing programs for images using reinforced adversarial learning. *arXiv preprint arXiv:1804.01118*, 2018.

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

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, volume 46, pp. 317–330. ACM, 2011.

Sumit Gulwani, William R Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105, 2012.

HoC. Hour of code. https://codehs.com/hourofcode/, 2018.

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

Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015.

Chengtao Li, Daniel Tarlow, Alexander Gaunt, Marc Brockschmidt, and Nate Kushman. Neural program lattices. In *ICLR*, 2017.

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

Richard E Pattis. *Karel the robot: a gentle introduction to the art of programming*. John Wiley & Sons, Inc., 1981.

Illia Polosukhin and Alexander Skidanov. Neural program search: Solving programming tasks from description and examples. *arXiv preprint arXiv:1802.04335*, 2018.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pp. 1139–1149, 2017.

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

Richard Shin, Illia Polosukhin, and Dawn Song. Towards specification-directed program repair. 2018.

Ashwin J Vijayakumar, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-guided deductive search for real-time program synthesis from examples. In *ICLR*, 2018.

Chenglong Wang, Kedar Tatwawadi, Marc Brockschmidt, Po-Sen Huang, Yi Mao, Oleksandr Polozov, and Rishabh Singh. Robust text-to-sql generation with execution-guided decoding. *arXiv preprint arXiv:1807.03100*, 2018.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Reinforcement Learning*, pp. 5–32. Springer, 1992.

Da Xiao, Jo-Yu Liao, and Xingyuan Yuan. Improving the universality and learnability of neural programmer-interpreters with combinator abstraction. In *ICLR*, 2018.

Xiaojun Xu, Chang Liu, and Dawn Song. Sqlnet: Generating structured queries from natural language without reinforcement learning. *arXiv preprint arXiv:1711.04436*, 2017.

Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. In *ACL*, 2017.

Wojciech Zaremba and Ilya Sutskever. Reinforcement learning neural turing machines-revised. *arXiv preprint arXiv:1505.00521*, 2015.

Wojciech Zaremba, Tomas Mikolov, Armand Joulin, and Rob Fergus. Learning simple algorithms from examples. In *Proceedings of The 33rd International Conference on Machine Learning*, pp. 421–429, 2016.

A MORE DESCRIPTIONS OF THE KAREL DOMAIN

Figure 4 presents the grammar specification of the Karel DSL.

Each Karel grid world has a maximum size of 18×18, and is represented as a $16 \times 18 \times 18$ tensor, where each cell of the grid is represented as a 16-dimensional vector corresponding to the features described in Table 4.

```
  Prog p  ::=  def run() : s
  Stmt s  ::=  while(b) : s | repeat(r) : s | s1 ; s2 | a
           |   if(b) : s | ifelse(b) : s1 else : s2
  Cond b  ::=  frontIsClear() | leftIsClear() | rightIsClear
           |   markersPresent() | noMarkersPresent() | not b
Action a  ::=  move() | turnRight() | turnLeft()
           |   pickMarker() | putMarker()
  Cste r  ::=  0 | 1 | ... | 19
```

Figure 4: Grammar for the Karel task.

Robot facing North
Robot facing East
Robot facing South
Robot facing West
Obstacle
Grid boundary
1 marker
2 markers
3 markers
4 markers
5 markers
6 markers
7 markers
8 markers
9 markers
10 markers

Table 4: Representation of each cell in the Karel state.

B MODEL DETAILS

B.1 NEURAL NETWORK ARCHITECTURE

Our neural network architecture can be found in Figure 1, which follows the design in Bunel et al. (2018). In particular, the IO Encoder is a convolutional neural network to encode the input and output grids, which outputs a 512-dimensional vector for each input-output pair. The decoder is a 2-layer LSTM with a hidden size of 256. The embedding size of the program tokens is 256.

Each program is represented as a sequence of tokens $G = [g_1, g_2, ..., g_L]$, where each program token g_i belongs to a vocabulary Σ. At each timestep t, the decoder LSTM generates a program token g_t conditioned on both the input-output pair and the previous program token g_{t-1}, thus the input dimension is 768. Each IO pair is fed into the LSTM individually, and we do a max-pooling operation over the hidden states of the last layer of LSTM for all IO pairs. The resulted 256-dimensional vector is fed into a softmax layer to obtain a prediction probability distribution over all the 52 possible program tokens in the vocabulary.

Notice that this neural network architecture can also be applied to other program synthesis problems, with modifications of the IO encoder architectures for different formats of input-output pairs. For

Ensemble	1	2	3	4	5	6	7	8	9	10
Shortest	39.40%	42.80%	43.56%	43.84%	44.32%	44.96%	45.16%	45.44%	45.52%	46.36%
Majority vote	39.40%	40.76%	41.56%	42.92%	42.84%	43.84%	43.68%	44.36%	44.48%	44.60%

Table 5: Exact match accuracy of the ensemble with our Exec training approach.

Ensemble	1	2	3	4	5	6	7	8	9	10
Shortest	86.04%	87.20%	88.64%	89.40%	89.64%	90.24%	90.32%	90.44%	90.48%	90.60%
Majority vote	86.04%	86.88%	88.24%	89.16%	89.20%	90.04%	90.08%	90.20%	90.56%	90.68%

Table 6: Generalization accuracy of the ensemble with our Exec training approach.

example, in the domain where input-output examples are text strings, such as FlashFill (Gulwani, 2011), the IO encoders can be recurrent neural networks (RNNs) (Devlin et al., 2017b).

B.2 Training Objective Functions

To estimate the parameters θ of the neural network, we first perform supervised learning to maximize the conditional log-likelihood of the referenced programs (Parisotto et al., 2017; Devlin et al., 2017b; Bunel et al., 2018). In particular, we estimate θ^* such that

$$\theta^* = \arg\max_{\theta} \prod_{i=1}^{N} p_\theta(\pi_i | \{IO_i^k\}_{k=1}^K) = \arg\max_{\theta} \sum_{i=1}^{N} \log p_\theta(\pi_i | \{IO_i^k\}_{k=1}^K) \tag{3}$$

Where π_i are the ground truth programs provided in the training set.

When training with reinforcement learning, we leverage the policy gradient algorithm REINFORCE (Williams, 1992) to solve the following objective:

$$\theta^* = \arg\max_{\theta} \sum_{i=1}^{N} \sum_{G} \log p_\theta(G | \{IO_i^k\}_{k=1}^K) R_i(G) \tag{4}$$

Where $R_i(G)$ is the reward function to represent the quality of the sampled program G. In our evaluation, we set $R_i(G) = 1$ if G gives the correct outputs for given inputs, and $R_i(G) = 0$ otherwise.

B.3 Training Hyper-parameters

We use the Adam optimizer (Kingma & Ba, 2015) for both the supervised training and the RL training. The learning rate of supervised training is 10^{-4}, and the learning rate of reinforcement learning is 10^{-5}. We set the batch size to be 128 for supervised training, and 16 for RL training.

C More Details of the Ensemble

For different training approaches of a single model, we train 10 models with different random initializations. To do the ensemble, we first sort the 10 models according to the descending order of their generalization accuracies on the validation set, then select the first k models to compute the results of the k-model ensemble. When multiple programs satisfy the ensemble criterion, e.g., with the shortest length for the *Shortest* method, we choose the one from the models with better generalization accuracies on the validation set.

Tables 5 and 6 show the numerical results of exact match and generalization accuracies of applying ensemble to our Exec approach respectively.