Predictive Synthesis of API-Centric Code

Daye Nam
Carnegie Mellon University†
U.S.A.

Baishakhi Ray
Columbia University†
U.S.A.

Seohyun Kim
Meta
U.S.A.

Xianshan Qu
Meta
U.S.A.

Satish Chandra
Meta
U.S.A.

Abstract

Today's programmers, especially data science practitioners, make heavy use of data-processing libraries (APIs) such as PyTorch, Tensorflow, NumPy, and the like. Program synthesizers can provide significant coding assistance to this community of users; however program synthesis also can be slow due to enormous search spaces.

In this work, we examine ways in which machine learning can be used to accelerate enumerative program synthesis. We present a deep-learning-based model to predict the sequence of API functions that would be needed to go from a given input to a desired output, both being numeric vectors. Our work is based on two insights. First, it is possible to learn, based on a large number of input-output examples, to predict the likely API function needed. Second, and importantly, it is also possible to learn to *compose* API functions into a sequence, given an input and the desired final output, without explicitly knowing the intermediate values.

We show that we can speed up an enumerative synthesizer by using predictions from our model variants. These speedups significantly outperform previous ways (e.g. DeepCoder [2]) in which researchers have used ML models in enumerative synthesis.

CCS Concepts: • **Software and its engineering → Programming by example**; Automatic programming; *API languages.*

Keywords: Program Synthesis, Programming By Example, PyTorch, Tensor Manipulation

† Work done at Facebook as an intern.
† Work done at Facebook as visiting scientist; equal contribution as the first author.

MAPS '22, June 13, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9273-0/22/06…$15.00
https://doi.org/10.1145/3520312.3534866

ACM Reference Format:
Daye Nam, Baishakhi Ray, Seohyun Kim, Xianshan Qu, and Satish Chandra. 2022. Predictive Synthesis of API-Centric Code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (MAPS '22), June 13, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3520312.3534866

1 Introduction

One of the cherished dreams of the programming languages research community is to enable automated synthesis of programs based on a specification. Synthesis approaches have been designed around several different forms of specification, e.g. a formal specification, [14, 15] or natural language description [30, 32], or input-output examples (aka demonstration) [10, 11, 22, 26, 31], or a combination thereof. Just as well, several different approaches to synthesis have been researched [13, 15, 27, 28].

Our focus is on coding assistance for users of numeric libraries such as PyTorch, Tensorflow, Numpy, Pandas, and the like, each of which provide powerful data manipulation routines behind an API, and the API functions are generally side-effect free. We assume a specification in the form of a single input-output example, and we are looking for a straight-line program consisting of calls to API functions. We choose *enumerative synthesis* as the underlying synthesis approach. Our research goal—shared with recent works such as DeepCoder [2], TF-Coder [23], and others—is to speed up plain enumerative synthesis using machine learning (ML).

Here is an input matrix as well as the desired output matrix, and the synthesis problem is to come up with a sequence of function calls that would convert the input to output. We will use the PyTorch API for this purpose.

```
in = [[5., 2.], [1., 3.], [0., -1.]]
out = [[[5., 5.], [1., 1.], [0., 0.]],
       [[2., 2.], [3., 3.], [-1., -1.]]]
```

The desired code fragment for this example is:

```
transpose(stack((in, in), 2), 0, 1)
```

The goal of program synthesis is to arrive at this expression, given only the input and output. Keep in mind that it is unlikely that random guessing of an expression will work: there are tens if not hundreds of available functions, and each function might take more than one argument. Thus, a systematic search is necessary.

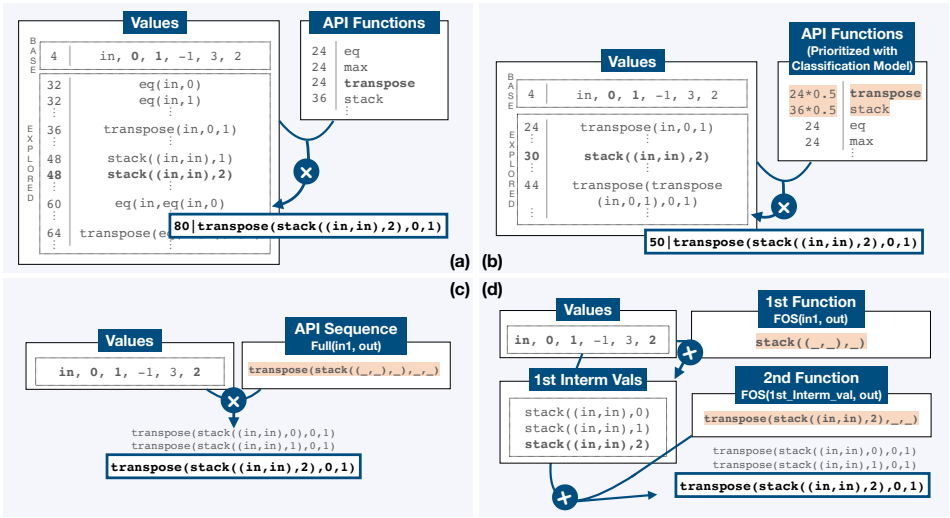

Figure 1. Overview of ML guided enumerative search algorithms. (a) weighted enumerative synthesis without ML model incorporation [23], (b) weighted enumerative synthesis with one-time ML-based prioritization [2, 23], (c) incorporation of Full-Seq prediction mode, (d) incorporation of First-Of-Seq prediction mode. Red-highlight indicates the API functions predicted by ML models. The underscore is a placeholder for argument values. Numbers (in (a),(b)) on the left side are the costs assigned to the values and API functions.

Basic Enumerative Synthesis. Refer to Figure 1, part (a), where we illustrate an enumerative synthesis in the style of Transit [28] and TF-Coder [23]. The idea is to organize the search in the order of increasingly complex expression trees, where the complexity is approximated by a *cost*. We heuristically assign a cost to each available value and to each operation, which here are API functions. At each step, we work with a budget, which grows in successive steps. Expressions that can be formed from existing values within the budget are added to a pool of values. For instance, the expression `stack((in,in),2)` would cost the two times the cost of `inp` plus the costs of the value `2` and the function `stack`. In the figure, this cost comes to 48, based on the cost of `stack` being 36. The value computed by this expression is added to the pool of values, along with the expression that computes them and its cost. The process continues until the desired output value is found.

Trying Likely Functions First. The enumerative search presented above is slow, and gets exponentially slower if a larger expression is needed to get the job done. A reason for this slowness is that the turn of the actually needed API function might come in quite late, as enumerative synthesis makes its way through the smaller cost budget and cheaper functions. Balog et al [2], in their seminal work DeepCoder, described a ML based strategy to accelerate enumerative program synthesis. DeepCoder's insight is to re-assign costs to functions—based on a ML model over the given input and output—such that the function(s) more likely to be needed in a *given* situation are prioritized. See part (b) of Figure 1. Here, given the specification, DeepCoder's ML model, adapted to our setting, correctly deems `transpose`, and `stack` as likely to be needed. Operationally, an enumerative synthesis process (e.g. like TF-Coder's [23]) can lower costs of these operations by some factor, so they are likely to be tried in preference to other API functions. The hope is that if the ML prediction is accurate, and the discounted costs work out, the process of enumerative synthesis can be sped up considerably.

This Paper: Predicting Function Sequences. Our thesis is that ML can be used in the setting of enumerative synthesis of API-centric code in a more powerful way: not for prioritization, but instead to directly predict the sequence of API functions that required to go from input(s) to the desired output. We describe two ways in which such a predictive model can be used to accelerate enumerative synthesis.

The first way in which we use this prediction model is to just let it predict the entire sequence of API functions in one shot, given the input and the output. In our running example, the model will predict `stack, transpose` as the sequence. See Figure 1, part (c). Given this sequence, the enumerative synthesizer will only look for values to fill into the function call arguments (shown by "_"). If the model predicts correctly, the search space that an enumerative synthesizer faces is vastly reduced, leading to possibly significant speedups.

A second way in which we use this predictive model is to use it as a "first-of-sequence" (FOS) predictor. See Figure 1, part (d). Given the input and the desired output, the FOS predictor only predicts the *first* function in the sequence needed. Say it predicts that function is `stack`. The synthesizer tries out a set of concrete arguments for `stack` from the values pool. The result of evaluating `stack` on each of

Table 1. Sample of synthesized programs with Full-Seq model guided enumerative synthesis and the synthesis time comparison. More examples can be found in the extended version [17].

Synthesized Program	Full-Seq (s)	no ML (s)
`eq(in1,unsqueeze(in1,1))`	0.18	0.8
`tensordot(in1,transpose(in2,0,1),1)`	0.32	2.06

these sets of arguments is added to the values pool; these are intermediate values in the desired computation. Next, for each intermediate value thus obtained, the synthesizer invokes the model again, this time giving it the intermediate value (in place of the input) and the desired output value. Say the model now predicts that the first function in the *remaining* sequence needed is `transpose`. The synthesizer then looks for appropriate arguments for `transpose`. At this point, one of the argument choices would provide the desired output. Compared to the full prediction, the point of this FOS mode is that it gets to predict on the basis on *known* intermediate values, a bit akin to teacher forcing [29] in sequence prediction, and can be successful more often than the full prediction mode; but it can be less efficient than one-shot prediction of the entire sequence.

On the running example, here are the comparative times to a successful solution: plain enumerative synthesis, 54.79 seconds; DeepCoder-style ML-based prioritization, 34.71 seconds; our API sequence prediction, FOS mode, 0.49 seconds; and API sequence prediction, Full mode, 1.45 seconds. In this example, full sequence prediction mode took a tad longer than the FOS mode: this is because the correct full sequence was in top-3 but not top-1, whereas in the FOS model the correct choice were at top-1. In general, we have found the full sequence mode to be faster than FOS mode. Other examples of Full-Seq guided synthesis are available in Table 1.

Contributions. We make two contributions in this work. First, we present a way to incorporate powerful predictive models in the context of enumerative program synthesis. On a suite of benchmarks (adapted from Stack Overflow) for PyTorch, using our ML models reduces the (mean, max) synthesis time from (10.01, 96.53) to (1.04, 9.58). By contrast, an adaptation of the idea of DeepCoder [2] reduces the (mean,max) synthesis time only to (7.44, 77.00). (See Section 5.3, Table 4.)

Second, our main technical advancement is in being able to carry out prediction of a sequence of API functions, given the input and the final desired output. Specifically, our model predicts one API function at a time and executes each predicted API function to convert the (intermediate) input state into another intermediate state until it becomes the target output state. Here, the intermediate states are not given to the model, but the model learns to represent what *would be* concrete intermediate values in the latent space during the training time. The ability to execute the API functions in the latent space indicates that the model learns the API function semantics (i.e., the relation between the input and output states) rather than the sequence distribution of the training dataset, and allows the model to generalize to unseen sequences or lengths. See Sec 6.

An extended version of this paper [17] contains additional details and datasets.

2 Learning to Predict API Sequences

Our technique works based on supervised learning over a large number of input/output examples, trained over individual API functions, or on sequences of API functions. Since the availability of real training data is a pervasive problem in ML, we use synthetic data generation similar to prior program synthesis work [25]. We pipe randomly-generated diverse inputs through sequences of API functions and collect resulting outputs (see Section 3.3). This helps capture the behavior of a single or a sequence of API functions in terms of how it transforms its input to the output.

Once trained, the model is able to predict a sequence of API functions. It can predict for input-output pairs that were never seen in the training data; thus it generalizes in the data space as long as the query input-output pairs are in distribution. More interestingly, it can predict sequences of API functions that were not seen in the training set either. This latter point is crucial, because the way we train the model, it learns to *compose* new, previously unseen sequences from the behaviors learned from training sequences.

Before we present operational details (Sec 3 onwards), we would like to present some intuitions behind our proposed ideas. We start with a basic classification model designed to predict one API function, given an input/output pair; and then build over it a compositional model that is designed to predict a sequence of API functions. The importance of examining the classification model on its own was crucial in our own journey, because it helped overcome several challenges in synthetic data generation for training. (In actual synthesis application, we use the model that predicts function sequences, described after this.)

Predicting a Function from Input-output Data. The first intuition we use is that for many common API functions, their behavior—the relationship of output to inputs—has simple patterns. Moreover, the behavior of a function is *discriminable* from behaviors of other functions based on simple clues. Many functions simply move around elements of a data structure (e.g. `transpose`) in easy to recognize patterns. In other cases, the operation is a simple element-wise computation. This suggests that a feed-forward neural network can be trained to predict likely functions—as in a multi-classification problem—from a representation of the input/output data. Such a network has to be trained on large amounts of input-output examples and their known (ground truth) functions.

We tested our intuition by training a model to classify among one of 33 PyTorch functions, using a synthetic dataset. The model achieved a test accuracy of 92.54%; moreover, a tSNE visualization (see [17], figure 2) corroborated that the network maps distinct functions to different regions of the latent space.

Predicting Sequences of Functions. The case of predicting an API *sequence* (e.g., `[stack, transpose]`) is harder. The intermediate values that flow between API calls are not known ahead of time, so it is not possible to reconstitute this sequence simply by invoking the classification model (for one API method name) over successive pairs of inputs and outputs. Moreover, learning to recognize the intended sequence from among all possible sequences, based on an input/output pair, can be difficult, for reasons for computational cost, for a classification model that predicts over a fixed collection of sequences of API function names.

This is where a second intuition comes into play. Given an input/output pair, we can imagine a model that predicts the *first* function in the intended sequence of the API functions that would process the input and eventually produce the (final) output. Crucially, we train this model as a *recurrent* unit, such that it not only predicts the first API function needed, but additionally produces a representation of the output of that first API function. This representation, along with the final output, can then be passed to a *recurrent* invocation of the model, to make it predict the next API function in the sequence. In this way, we can train a compositional model for API sequences.

In our running example, the model first predicts `stack` based on `inp` and `out`. It also computes an internal representation of the intermediate value `stack((inp,inp),2)`. It then predicts `transpose` based on this internal representation and `out`. This is the principle by which the model is able to compose even longer previously-unseen sequences.

Here we emphasize that the model is *not* predicting the next API token (e.g. `transpose`) based on the tokens that came before (e.g. `stack`), as is done in code completion models [12]. At each step, the prediction is based *only* on (a representation of) the program state, as opposed to on program text. This is a new capability, which could be combined with additional signals such as previous tokens, if desired.

3 Technical Details

In this section, we explain our models. We will use Figure 2 to show details using an example.

3.1 The *Encoding* Function

Before passing the input/output tensors to the models, we encode them into a fixed-length vector (Figure 2-Encoding). We extract three different pieces of information from the tensors: (i) tensor values, (ii) tensor shapes, and (iii) tensor types, and combine them as a sequence separated by a special separator $< s >$, *i.e.,* X = type <s> shape <s> value, such that, the models can learn from all the three modalities together.

To manage the wide range of tensor values in the model, we normalize the values as follows: we encoded the values greater than 100 into 100, values greater than 1000 into 101, and similarly for the negative values. The intuition is based on how developers recognize patterns: when a value becomes large enough, the importance of the least significant digit decreases in pattern recognition.

Finally, all domain inputs and output encoding are concatenated together. We support up to 3 inputs and one output. Dummy inputs are added when there are less then 3 inputs to keep the model input size same for all examples.

3.2 Compositional Model

We train a model to predict the sequence of API functions $s_f = [f_1, ..., f_n]$, given a task specification $\phi = \{inp, out\}$, where *inp* is a list of input tensors that have gone through s_f, and *out* is the final output tensor.

Figure 2 Compositional Model shows three units of the model. In each unit, the encoding passed to the feed forward network is similar to the one used before to create an embedding of an input-output pair. When f_i needs to use $f_{i-1}(args_{i-1})$, we mask the position as empty ("<p>") so that the model exploits $\mathbf{h}_{i-1}$. Embedded encodings are passed to RNN units, and each unit further projects the input embedding into the RNN embedding space to generate h_i, using information flowed from adjacent units, h_{i-1}. Finally, the output of each unit is passed to a softmax layer (not shown here) to produce a probability distribution over API functions.

3.3 Synthetic Data Generation

To train a neural model so that it can understand the behavior of API functions, a large number of corresponding input-output pairs is necessary. Unlike other problems exploiting ML models, collecting real-world data from code repositories (e.g., GitHub) is not applicable here because we need runtime values, not static information such as static code. Therefore, we randomly generate input/output values, and use the synthetic dataset for model training.

Listing 1. Example data generation code for `torch.sum`

```
def generate_sum_IO():
    in_tensor, tensor_size = random_tensor()
    dim = random_dimension(0, tensor_size)
    if dim == len(tensor_size):
        out_tensor = torch.sum(in_tensor)
    else:
        out_tensor = torch.sum(in_tensor, dim)
    return (in_tensor, out_tensor)
```

For each API function, we randomly generate input tensors, run the API functions with them, and capture the corresponding outputs. In other words, we create a set of input/output values in a black-box manner: we do not assume

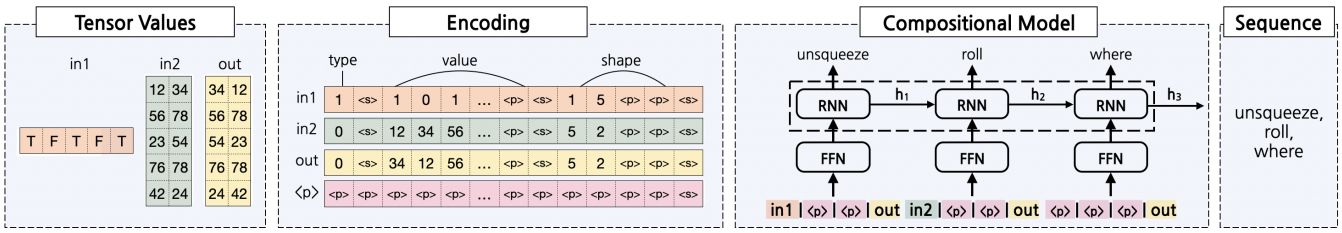

Figure 2. Illustration of Compositional Model on an example. The inputs are in the Tensor Values box, and the expected prediction is shown in the Sequence box.

API functions' implementation details. As it does not require understanding internal program structures, it is easy to generate a large number of input-output pairs without much manual effort and can be easily parallelized.

However, as even a simple API operation in modern libraries (e.g., PyTorch) imposes many constraints, inputting random values will generate many runtime errors due to the constraints violations. To reduce such errors and generate meaningful input/output data points efficiently, we exploit API specification, and generate a set of inputs with the valid combinations (see Listing 1 for an example).

4 Incorporating ML in Enumerative Syn.

Here we formally describe how the ML models were incorporated into the enumerative synthesis. Please refer to Figure 1 and Section 1 for a walk through of these on an example. Detailed description of our implementation and the pseudo code for each synthesis approach can be found in the supplementary material (Appendix C, D).

Basic Enumerative Synthesis. As a baseline, we implement an enumerative synthesizer without any ML models. Basic enumerative search starts with a set of base values and enumerates over combinations of operations and the values.

The list of base values includes `inp`, other basic constants such as `0`, `1`, `-1`, or heuristically-chosen values such as the dimensions of the given variables (e.g., `3`). Then, starting with the base values, the search enumerates ways of applying operations to previously-explored values and expand the set of known values. There are various ways of iterating the operations and the values, (e.g. based on syntactic size as in Transit [28]), but we use weighted enumerative search, which is the approach of and TF-Coder [23]. It does so in the order of increasing *cost*. Operations and values are assigned costs based on their complexity: less common and more complex operations are assigned higher cost. Costs are additive, so common operations and simpler expressions are explored earlier. The costs are manually set by the synthesizer developers once, and it will be used for all tasks.

Prioritizing Likely Functions with an ML Model. As the needed operations for a specific problem are not known to the synthesizer ahead of time, the costs seeded in it will not always be ideally suited for all problems. TF-Coder [23] and DeepCoder [2] address this problem using an ML model to re-weigh all operations before the enumerative search starts. Given input/output examples, it invokes a multi-label classification model to predict the probability of each needed operation and re-weighs them accordingly with the goal of encountering the needed operations earlier in the search. We trained classification model following DeepCoder [2].

Compositional Model - Full-Sequence. In this mode, compositional model predicts a sequence of API functions $s_f = [f_1, f_2, ..., f_n]$ given the final output *out*, and the inputs to each API function $[in_1, in_2, .., in_n]$. The synthesizer invokes the compositional model with the specification, predicts a sequence of operations, and searches only the parameter values (e.g., *dimension*) that were not provided in the specification.

The Full-Seq mode completely bypasses enumerative search over operations. Instead, the compositional model predicts the API functions needed in a synthesis instance as well as the order of those APIs in the synthesized code. Thus, the synthesizer does not need to search the operation space, but only needs to search the combinations of base values.

Compositional Model - First-Of-Sequence. In the First-Of-Seq mode, given an input/output pair, compositional model predicts the most probable API function needs to come in the sequence. As enumerative search keeps track of the intermediate output value, we can iteratively invoke compositional model, and compute the intermediate values using the predicted API functions, which can be used to predict the next API function.

5 Evaluation

In this section, we first describe the dataset (Section 5.1), and evaluate the trained API function sequence prediction model (Section 5.2). Then, we investigate the prediction-guided synthesis (Section 5.3). Finally, we show the generalizability and the compositional property of our model (Section 5.4).

5.1 Dataset

Program Synthesis Benchmarks. We evaluated the effectiveness of our approaches with a subset of TF-Coder's SO benchmarks [23]. These benchmarks contain 50 tensor manipulation examples collected from SO, each containing

Table 2. Statistics of the dataset used in this study. Numbers in parentheses indicate the length of the sequences.

	Synthetic			Stack Overflow
	Train	Valid.	Test	Test
# of unique seqs (len)		16 (1) + 186 (2)		8 (1) + 7 (2)
# of in/out values	5.5M	10K	10K	18

input/output tensor values and the desired solutions in Tensorflow. To evaluate our approach that supports PyTorch, we first translated them into PyTorch and excluded tasks that we could not translate by hand. Among the 33 API functions needed to for remaining 36 benchmarks, we selected 16 functions covering 18 benchmarks (Table 2-Stack Overflow) from core utility that modify values (e.g., `add`) or shapes (e.g., `transpose`) of tensors, create them, or manipulate them in similar ways. These operations were chosen because the model can clearly observe the behavior of each API function solely from input/output pairs (i.e., no side effects). The full benchmarks we support are available in the extended version [17].

Synthetic Data Generation. To train our sequence prediction model working for the SO benchmarks, we synthesized a dataset as per Section 3.3. We synthesized 202 unique sequences by using the exhaustive combination of 16 API functions, with 1 or 2-length sequences. From the 272 ($16 + 16^2$) possible sequences, 70 were removed due to the constraints.

For each API function sequence in the training dataset, say f_1, f_2, f_3, we ran f_1 with randomly generated input and other parameter values (e.g., dimension). Then, f_2 takes f_1's output as input and takes other random input tensors, if necessary. We treat f_3 similarly by propagating f_2's output.

It took 1-person week to encode the API specifications to write valid data generation code by reading the PyTorch documentation. To avoid expansion to large input values and to let the model learn the patterns sufficiently, we used a fixed range of values (from 0 to 20) and the size of tensors (up to 3 dimensions, and up to 5 elements in each dimension), to prevent the tensors to be dispersed too much.

We created the dataset with 100K input/output pairs for each unique API sequence (Table 2-Synthetic), and split it into training, validation, and test sets. Each included all 202 API sequences, but the input/output values were not overlapped across the datasets.

5.2 Sequence Prediction Model

We trained both Full-Seq and First-Of-Seq variants using the training set of the synthetic data, and evaluated it with (1) the test set of the synthetic data, and (2) SO benchmarks.

Observation. Table 3 shows the result. Model's top-1 testing accuracies of the 10K synthetic test set are ~79%. Among 18 SO benchmarks, the Full-Seq model found 13 sequences

Table 3. Model accuracy for unseen input/output values.

Model	Synthetic-Test Top-1	Stack Overflow Top-1	Stack Overflow Top-3
Full-Seq	79.36%	35.29%	76.47%
First-Of-Seq	66.88%	52.38%	76.19%

Table 4. End-to-end program synthesis results; our models in **bold**. Time, Max, and Median show the average, max, median synthesis time of found programs.

			Time		
	Found	Not Found	Mean	Max	Median
Enumerative	18	0	10.01	96.53	0.46
Multi-label	18	0	7.44	77.00	0.32
First-Of-Seq	17	1	5.87	59.93	0.39
Full-Seq	14	4	1.04	9.58	0.25

are in top-3 (72.22%), among them 6 are in top-1 (33.33%). In comparison to the Full-Seq model, the First-Of-Seq model's top-1 accuracy is better. This is not surprising as First-Of-Seq model has more information (actual values of the intermediate inputs) than the Full-Seq variant. However, surprisingly, the top-3 accuracies of both are almost similar. These results indicate that the Full-Seq model perhaps learned a representation of the intermediate states of the API sequence: even without passing the true intermediate values, the Full-Seq model behaves at per with the Full-Seq model at top-3.

5.3 Prediction-guided Enumerative Synthesis

Using the trained Full-Seq and First-Of-Seq variants, we first evaluate our approach, against vanilla enumerative synthesis (similar to [23]). We further compared with a DeepCoder [2]-like multi-label prediction model. Table 4 shows the results.

Existing Synthesizers vs. Compositional Model. Among the 18 tasks, both vanilla enumerative search and a synthesizer prioritized with multi-label classification model could synthesize all tasks. The new variants incorporating our models synthesized 17 (First-Of-Seq) and 14 (Full-Seq) correctly, out of 18 and 17 respectively. However, although they synthesized fewer solutions, they required less time to synthesize the solutions: 5.87 seconds (First-Of-Seq) and 1.04 seconds (Full-Seq) on average, whereas the existing synthesizers took 10.01 and 7.44 respectively.

We see this speed up because predicting the sequence reduces the search space. As the compositional models return a sequence of API functions, the enumerative search can focus on the argument values instead of iterating over the API function sequences. Note that the multi-label classification model also suggests potential API functions, it provides us with a set of functions, not sequences. Thus, in the worst case, the associated enumerative search has to explore all the possible combinations increasing the synthesis time.

The difference in synthesis time between the compositional models and the baselines is not big in simple tasks (e.g., `any(in,-1)`), which is why the difference in median in Table 4 is not significant enough. However, when it comes to more complex tasks like in Figure 1, the difference becomes significant: 54.79 seconds with plain enumerative synthesis vs. 0.49 with First-Of-Seq mode.

One caveat of compositional models is that the model prediction is not the bottom-up method but a one-shot approach. Therefore, when the model fails to predict the sequence correctly, it cannot synthesize the program. However, as the whole search can be done quickly, the time overhead is not high even when one tries with the compositional model and employ other approaches once it fails.

First-Of-Seq vs. Full-Seq. Between the two variants, First-Of-Seq was able to synthesize more programs. For example, First-Of-Seq successfully synthesized the desired program `where(lt (in,1),in,1)` by predicting `lt` and `where` correctly in top-3. However, Full-Seq failed, and predicted `[[eq, where],[eq,mul],[gt,where]]` as top-3, which are close, but not entirely correct. This is expected. First-Of-Seq only has to get the first element of the sequence right; the next element is in fact the first element of the result of the *subsequent* prediction, which in turn, is based on the actual intermediate value computed by the first function predicted. Whereas, the Full-Seq gets only one chance to get the entire sequence right without knowing the intermediate values. When a sequence is correctly predicted, Full-Seq model could synthesize the solutions faster; it only needs to be invoked once, without the intermediate values computation.

5.4 Evaluating Generalization

To see whether the model truly learned the functionality of the API functions and learned their compositions, we tested whether the model can generate new API sequences that were not present in the training data. From the original synthetic dataset 2, we removed the data of 7 API function sequences with length-2 that were included in SO benchmarks, and trained the First-Of-Seq and Full-Seq models. Our hypothesis was that if the models are able to learn the compositional property, instead of learning the distributions of sequences, they should be able to generate unseen sequences by composing API functions into a sequence.

Observation. Not surprisingly, the accuracy drops from the Section 5.2 result. Nevertheless, out of 8 benchmarks with 2 sequence, we can still predict 4 sequences at top-5 (50% accuracy) with Full-Seq, and 6 sequences (75% accuracy) with First-Of-Seq. In this setting, we sometimes narrowly miss some function sequences. For example, we miss a benchmark [`lt`, `where`], however it predicts [`eq`, `where`], and [`gt`, `where`] instead. Note that `gt` and `eq` have very similar functionalities to the intended API function `lt`. Both the models can correctly predict sequences like [`unsqueeze`, `eq`], [`matmul`, `add`], etc. As expected, the First-Of-Seq model works much better than Full-Seq model.

Table 5. Model accuracy for unseen input/output values, trained with a dataset covering all SO benchmarks.

Model	Synthetic-Test Top-1	Stack Overflow Top-1	Stack Overflow Top-3
Full-Seq	88.15%	68.57%	91.42%
First-Of-Seq	65.44%	51.61%	79.03%

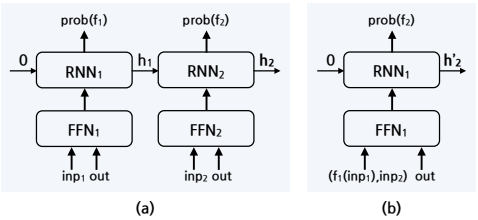

Figure 3. Illustration of compositional learning. (a): Two units of the compositional model predicting a sequence $[f_1, f_2]$. (b): Single unit model predicting f_2 given $f_1(in_1)$ instead of h_1. We show the compositional property of the model by showing $h_2 \approx h'_2$.

To further check the model's ability to generalize to unseen 3-length sequence, we randomly picked 71 unique 3-length sequences made out of 16 API functions and collected 100 instances of them with different input/output values. This gives a total of 7100 test samples. We used the model trained with only sequences with length 2. Overall, at top-5, model's accuracy is ~34% when queried with unknown sequences and unknown values. However, the model can predict 69 out of 71 sequences correctly at least with one input/output. The only two sequences the model missed are [`add`, `mul`, `any`] and [`add`, `unsqueeze`, `ne`]. In contrast, [`where`, `expand`, `matmul`] was predicted correctly around 97% time. These results indicate the model's ability to generalize.

6 Why Composition Works?

We show that a unit of our compositional model has an interesting property: it learns to convert its incoming hidden vector to its outgoing hidden vector in a way consistent with the *semantics* of the API function it predicts, albeit in embedding space. This property is crucial for predicting a sequence compositionally.

In Fig 3(a), we show two units of the compositional model, where the first one predicts function f_1, on the basis of inp_1, *out* and the previous hidden vector, if any. That unit also produces a hidden vector h_1. The second unit produces hidden vector h_2. It may also consume further local input (such as inp_2). Fig 3(b) shows an alternate situation in which we give the *result* of $f_1(in_1)$ directly as input to the first unit, which then produces h'_2.

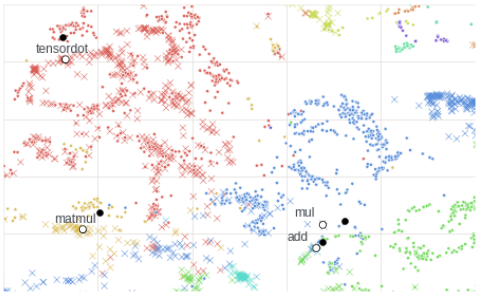

Figure 4. Proximity of h_2 and h_2' pairs for some inputs (in white and black respectively), against a backdrop of h_2 (crosses) and h_2' (dots).

The interesting property is that h_2 and h_2' are close together in the representational space. In Figure 4, as we expect, black and white markers show that $h2$ and $h2'$ for the same inputs are arranged close to each other in the embedding space. In producing h_2 (or h_2'), the RNN unit did not care whether it was given h_1, the representation produced by the previous RNN unit, or directly given $f_1(in_1)$. In this manner, successive hidden states contain information analogous to the results of concrete computations: $f_1(in_1)$, $f_2(f_1(in_1))$, and so on. (In the actual model, these functions need not be unary, as implied here.)

7 Limitations

Our results are promising, yet preliminary in many ways, and we have not established generality in several dimensions. First, we support a small set of API functions and have carried out a limited evaluation. As the number increases, the training data size also increases, and training the model well becomes harder due to computational needs. The robustness of training is a challenge in general.

Second, the model's ability to generalize to unseen sequences is crucially dependent on training over a broad diversity of API sequences. This is challenging as we go to a larger number of API functions, because we cannot cover all permutations exhaustively. However, the model can still learn the semantics reasonably well if the training data covers the sequences in the test set. Thus, the future works may benefit from creating a training dataset containing a distribution of sequences representing the real-world API usage patterns, through API usage mining [16, 34].

Third, we have explored the model's training and inference on relatively short tensors, with small data ranges, and have generally worked only with integer data. In a real application, tensors can be out-of-distribution.

Fourth, we have worked only with PyTorch. We believe the work can be replicated easily to NumPy and Tensorflow API functions, because of their similar nature (acting over arrays of numbers.). Farther out, we may need to invent additional techniques.

8 Related Work

ML for Program Synthesis. With advances in ML, researchers tried to adopt ML on top of the enumerative search for more efficient program synthesis [2, 4, 18, 20, 23]. Our work is closely related to DeepCoder [2], TF-Coder [23] and BUSTLE [20]. Using prediction-guided enumerative synthesizer, they show the benefits of predicting API functions that are needed *somewhere* given a synthesis instance. However, they all use a explicit featurization over these the input-output values, which is not easy to generalize to other programming languages. Also, they only predict presence or absence of API functions, the prediction was only used to prioritize operations in the enumerative search, rather than directly predicting the API function(s) in sequence. With the ML model guiding the search, BUSTLE takes an approach similar to ours, which gives feedback to search iteratively, whereas the models of DeepCoder or TF-Coder only give feedback in the beginning of the search. However, BUSTLE and DeepCoder only support simple DSL tasks, which may not be generalized for real-world API-based synthesis.

Neural Program Synthesis. Approaches like [1, 3, 5–9, 19, 21, 21, 24, 33] directly use neural networks for end-to-end synthesis [3, 5, 8, 21] to generate string transformation programs from examples. These works generally use encoder-decoder model. In particular, the encoder embeds the input/output strings, and the decoder generates the program sequences conditioned on the input embedding. However, these approaches are mostly built and evaluated with simple DSL tasks, mostly with simple string transformation. In this work, we worked on the real-world tensor manipulation library PyTorch. Although our evaluation does not cover the full range of PyTorch, we found several challenges in expanding these work into more complex programs, such as the scalability issue in training data generation and diversity of the API parameters especially in the tensor domain.

9 Conclusion

In this paper, we proposed a new machine learning technique to speed up enumerative program synthesis. Our idea is to use an ML model to predict the sequence of API function calls required to go from an input to the final desired output, in our case, both numeric vectors. Our model is trained on randomly generated data. It is able to predict API sequences for previously unseen inputs and outputs. Moreover, it can predict API sequences that were not seen during training either. The model does so by learning to compose API sequences, by learning how to keep track of values in the hidden states of an RNN. We showed that our model can predict sequences of lengths 1 to 3 fairly well. In terms of effectiveness, we showed that our technique accelerates enumerative synthesis more effectively than related previous works DeepCoder [2] and TF-Coder [23].

References

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *2021 Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*.

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

[3] Matej Balog, Rishabh Singh, Petros Maniatis, and Charles Sutton. 2020. Neural program synthesis with a differentiable fixer. *arXiv preprint arXiv:2006.10924* (2020).

[4] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. AutoPandas: neural-backed generators for program synthesis. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–27.

[5] Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. 2018. Leveraging grammar and reinforcement learning for neural program synthesis. *arXiv preprint arXiv:1805.04276* (2018).

[6] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Execution-guided neural program synthesis. In *International Conference on Learning Representations*.

[7] Xinyun Chen, Dawn Song, and Yuandong Tian. 2021. Latent execution for neural program synthesis beyond domain-specific languages. *Advances in Neural Information Processing Systems* 34 (2021).

[8] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*. PMLR, 990–998.

[9] Kevin M Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Joshua Tenenbaum, and Armando Solar-Lezama. 2019. Write, execute, assess: Program synthesis with a repl. (2019).

[10] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. *ACM SIGPLAN Notices* 52, 6 (2017), 422–436.

[11] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330.

[12] Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the Naturalness of Software. 59, 5 (2016).

[13] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. IEEE, 215–224.

[14] Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. Denali: A goal-directed superoptimizer. *ACM SIGPLAN Notices* 37, 5 (2002), 304–314.

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

[16] Daye Nam, Amber Horvath, Andrew Macvean, Brad Myers, and Bogdan Vasilescu. 2019. Marble: Mining for boilerplate code to identify API usability problems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 615–627.

[17] Daye Nam, Baishakhi Ray, Seohyun Kim, Xianshan Qu, and Satish Chandra. 2022. Predictive Synthesis of API-Centric Code. *CoRR* abs/2201.03758 (2022). arXiv:2201.03758 https://arxiv.org/abs/2201.03758

[18] Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. 2019. Learning to infer program sketches. In *International Conference on Machine Learning*. PMLR, 4861–4870.

[19] Maxwell Nye, Yewen Pu, Matthew Bowers, Jacob Andreas, Joshua B Tenenbaum, and Armando Solar-Lezama. 2020. Representing Partial Programs with Blended Abstract Semantics. *arXiv preprint arXiv:2012.12964* (2020).

[20] Augustus Odena, Kensen Shi, David Bieber, Rishabh Singh, Charles Sutton, and Hanjun Dai. 2020. BUSTLE: Bottom-Up program synthesis through learning-guided exploration. *arXiv preprint arXiv:2007.14381* (2020).

[21] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2016. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855* (2016).

[22] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 404–415.

[23] Kensen Shi, David Bieber, and Rishabh Singh. 2020. TF-Coder: Program Synthesis for Tensor Manipulations. *arXiv preprint arXiv:2003.09040* (2020).

[24] Eui Chul Shin, Illia Polosukhin, and Dawn Song. 2018. Improving neural program synthesis with inferred execution traces. *Advances in Neural Information Processing Systems* 31 (2018).

[25] Richard Shin, Neel Kant, Kavi Gupta, Christopher Bender, Brandon Trabucco, Rishabh Singh, and Dawn Song. 2019. Synthetic datasets for neural program synthesis. *arXiv preprint arXiv:1912.12345* (2019).

[26] Calvin Smith and Aws Albarghouthi. 2016. MapReduce program synthesis. *Acm Sigplan Notices* 51, 6 (2016), 326–340.

[27] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. 404–415.

[28] Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo MK Martin, and Rajeev Alur. 2013. TRANSIT: specifying protocols with concolic snippets. *ACM SIGPLAN Notices* 48, 6 (2013), 287–296.

[29] Ronald J Williams and David Zipser. 1989. A learning algorithm for continually running fully recurrent neural networks. *Neural computation* 1, 2 (1989), 270–280.

[30] Frank F Xu, Bogdan Vasilescu, and Graham Neubig. 2021. In-ide code generation from natural language: Promise and challenges. *arXiv preprint arXiv:2101.11149* (2021).

[31] Navid Yaghmazadeh, Xinyu Wang, and Isil Dillig. 2018. Automated migration of hierarchical data to relational tables using programming-by-example. *Proceedings of the VLDB Endowment* 11, 5 (2018), 580–593.

[32] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: query synthesis from natural language. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–26.

[33] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vol. 1. 440–450.

[34] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. 2018. Are code examples on an online q&a forum reliable?: a study of api misuse on stack overflow. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 886–896.