AIN'T NOBODY GOT TIME FOR CODING: STRUCTURE-AWARE PROGRAM SYNTHESIS FROM NATURAL LANGUAGE

Anonymous authors
Paper under double-blind review

ABSTRACT

Program synthesis from natural language (NL) is practical for humans and, once technically feasible, would significantly facilitate software development and revolutionize end-user programming. We present SAPS, an end-to-end neural network capable of mapping relatively complex, multi-sentence NL specifications to snippets of executable code. The proposed architecture relies exclusively on neural components, and is built upon a tree2tree autoencoder trained on abstract syntax trees, combined with a pretrained word embedding and a bi-directional multi-layer LSTM for NL processing. The decoder features a doubly-recurrent LSTM with a novel signal propagation scheme and soft attention mechanism. When applied to a large dataset of problems proposed in a previous study, SAPS performs on par with or better than the method proposed there, producing correct programs in over 90% of cases. In contrast to other methods, it does not involve any non-neural components to post-process the resulting programs, and uses a fixed-dimensional latent representation as the only link between the NL analyzer and source code generator.

Keywords: Program synthesis, tree2tree autoencoders, soft attention, doubly-recurrent neural networks, LSTM, nlp2tree.

1 INTRODUCTION

Program synthesis consists in automatic or semi-automatic (e.g., interactive) generation of programs (or other executable structures) from specifications. This task can be posed in several ways. It is most common to assume that specification has the form of input-output pairs (tests), in which case synthesis resembles learning from examples. A popular approach to this class of tasks is genetic programming.

Synthesis from examples is limited by its inductive nature: even if a program passes all provided tests, little can be said about generalization for other inputs. This is one of the rationales for synthesis from formal specifications, which are typically expressed as a pair of logical clauses (a *contract*): a precondition that constraints the set of acceptable inputs, and a postcondition that defines the properties one requires from program output. Programs so synthesized are correct by construction, but the task is NP-hard, so producing programs longer than a few dozen of instructions becomes computationally challenging. Synthesis from formal specifications is also difficult for programmers, because writing a specification of a nontrivial program can be actually *harder* than its implementation.

From a practical perspective, the most intuitive and convenient way of specifying programs is natural language (NL). This way of formulating synthesis tasks has been rarely studied in the past, when it was beyond the capabilities of available methods. The recent progress in NLP and deep neural networks made it more realistic, as confirmed by a few studies we review in Section 5. Here, we propose Structure-Aware Program Synthesis (SAPS[1]), an end-to-end approach to program synthesis from natural language. SAPS receives a short NL description of requested functionality and produces a snippet of code in response. To that aim, we combine generic word and sentence embeddings with tree2tree autoencoders trained on abstract syntax trees (ASTs). The unique feature of our

[1]Pun intended.

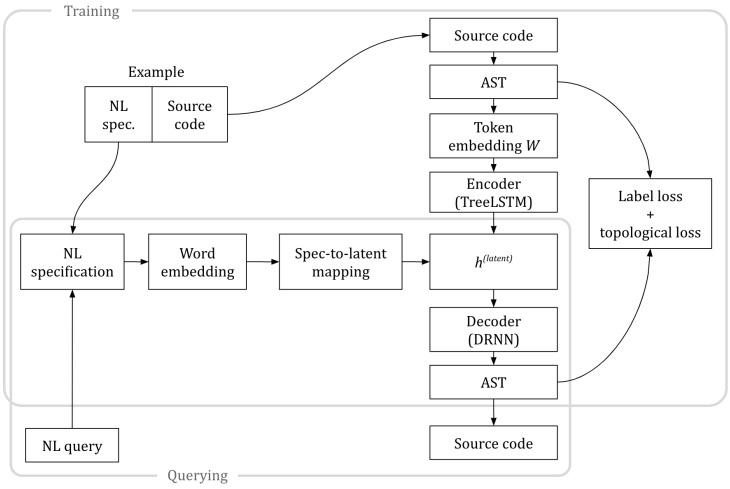

Figure 1: Overall architecture of SAPS.

approach is that the entire NL specification is folded into a single point in a fixed-dimensional latent space, which is then mapped by decoder onto the AST of a code snippet. This modular architecture facilitates usage of pretrained components, e.g. GloVe word embeddings (Pennington et al., 2014). We also enhance the original tree2tree architecture and augment it with global soft attention mechanism. Last but not least, the approach engages a form of dynamic batching for efficient learning from tree structures.

2 SAPS ARCHITECTURE

SAPS accepts a short NL specification as input and produces in response an AST of a snipped of code (program). It comprises of three main components (Fig. 1): (i) a word embedding for preprocessing of the specification, (ii) a mapping from the embedded sequence of specification tokens to a latent representation $h^{(latent)}$, and (iii) a decoder that unfolds $h^{(latent)}$ into an AST. The decoder comes from a tree2tree autoencoder trained via autoassociation on ASTs (the vertical sequence of blocks in the figure). The modules are largely independent, the only requirement being that (ii) can be trained only once (i) and (iii) are known. Therefore, we describe them in that order in the sections that follow.

2.1 WORD EMBEDDING

Inspection of the NL specifications proposed in Polosukhin & Skidanov (2018), which we use in the experimental part, revealed that they largely use standard English vocabulary, with only occasional occurrence of terms typical for programming. Given that, we rely on a pretrained GloVe embedding, more specifically the Common Crawl, which has been trained on generic NL on 42B tokens by Pennington et al. (2014), has vocabulary size of 1.9M tokens, and embeds the words in a 300-dimensional space. We chose this particular embedding, because the vocabularies of the other ones shared by Pennington et al. (2014) did not cover all terms occurring in the considered dataset.

Given an input NL query phrase of n tokens, this module produces a sequence of n 300-dimensional vectors $(q_1, \ldots, q_n)$. In order to be able to train the network with mini-batches of data, we pad each sequence in the mini-batch with a special out-of-vocabulary value, *PAD*.

2.2 LATENT-TO-AST MAPPING

We learn a latent, fixed-dimensionality embedding space of code snippets by autoassociative training of a tree-to-tree autoencoder, $enc \circ dec$, where $enc : X_{AST} \rightarrow X_l$, $dec : X_l \rightarrow X_{AST}$, X_{AST} is the space of programs' ASTs, and X_l is the latent space. Once trained, the encoder enc is scrapped, and we use dec as the latent-to-AST mapper.

Our autoencoder architecture, which we originally introduced in (Anonymous, 1999), is based on Tai et al. (2015) and Alvarez-Melis & Jaakkola (2017). However it significantly diverges from those works in using only the latent vector for passing the information between encoder and decoder.

Encoder. A single training example is $(x, x) \in X_{AST} \times X_{AST}$, where x is an AST of a snipped of code. Given (x, x), enc starts by encoding the label of every node x_j in x using one-hot encoding $v_j = V(x_j)$; for the experiments presented in Section 4, there are 72 node labels, so $|v_j| = 72$. Then, following Mikolov et al. (2013), we cast v_js, for each node in x independently, to a lower-dimensional space, using a learnable embedding matrix W: $y_j = Wv_j$. The result is a tree y of node label embeddings y_j, isomorhpic to x.

The encoder network folds the tree bottom-up, starting from the tree leaves, aggregating the hidden states h_j, and ending up at the root node. Because the arity of tree nodes may vary, we apply a recurrent TreeLSTM network, i.e. a variant of LSTM, to merge the information from children nodes (descendants). A TreeLSTM takes into account not only a single preceding node, but all descendant nodes.

The hidden states are initialized with zeroes. For each node, we first compute the sum of the hidden states of its descendants (children) $C(j)$:

$$\widetilde{h_j} = \sum_{k \in C(j)} h_k \tag{1}$$

For leaf nodes, $\widetilde{h_j} = 0$. Then we compute the values of input, forget and output gates:

$$i_j = \sigma(W_i y_j + U_i \widetilde{h_j} + b_i) \tag{2}$$

$$f_{jk} = \sigma(W_f y_j + U_f h_k + b_f) \tag{3}$$

$$o_j = \sigma(W_o y_j + U_o \widetilde{h_j} + b_o) \tag{4}$$

$$u_j = \tanh(W_u y_j + U_u \widetilde{h_j} + b_u) \tag{5}$$

where σ is the sigmoid function, and W_i, W_f, W_o and W_u and b_i, b_f, b_o and b_u are, respectively, learnable weights and biases. Note that the values of i_j, o_j and u_j depend on the aggregated children's states ($\widetilde{h_j}$), whereas f_{jk} is computed child-wise, yielding a separate output f_{jk} of forgetting gate for each child of the current node.

We then update the state c_j of the memory cell of the current node:

$$c_j = i_j \odot u_j + \sum_{k \in C(j)} f_{jk} \odot c_k \tag{6}$$

where $\odot$ stands for elementwise multiplication, and where we assume that the initial states of c_js of nonexisting children of leaves are zero. Next, we compute the hidden state of the current node:

$$h_j = o_j \odot \tanh c_j, \tag{7}$$

which is then recursively aggregated by the above procedure. Once this process reaches the root node, the obtained state h_j becomes the latent vector, i.e. $h^{(latent)}(x) = enc(x)$. Additionally, we have applied *layer normalization* (Ba et al., 2016) in the same manner as to traditional LSTM.

Decoder. Our decoder is a doubly recurrent neural network (DRNN) proposed in Alvarez-Melis & Jaakkola (2017), with modified scheme of state propagation and novel attention mechanism applied to the latent vector (see Fig. 2). DRNN comprises two LSTM cells, which separately capture the *vertical* and *horizontal* traversal of the resulting AST tree. The state of the former (h_i^a) is passed vertically downwards from parents to children, and of the latter (h_i^f) horizontally left-to-right to consecutive siblings. Importantly, in the original DRNN Alvarez-Melis & Jaakkola (2017), the state of the horizontal LSTM was reset before processing the children of each node. In contrast, we always pass it between all nodes the same tree depth – even between those that are not siblings – as well as between the last node at current depth and the first node at the next depth level. Consequently, the horizontal LSTM works in breadth-first order, while the vertical one in depth-first mode, as shown

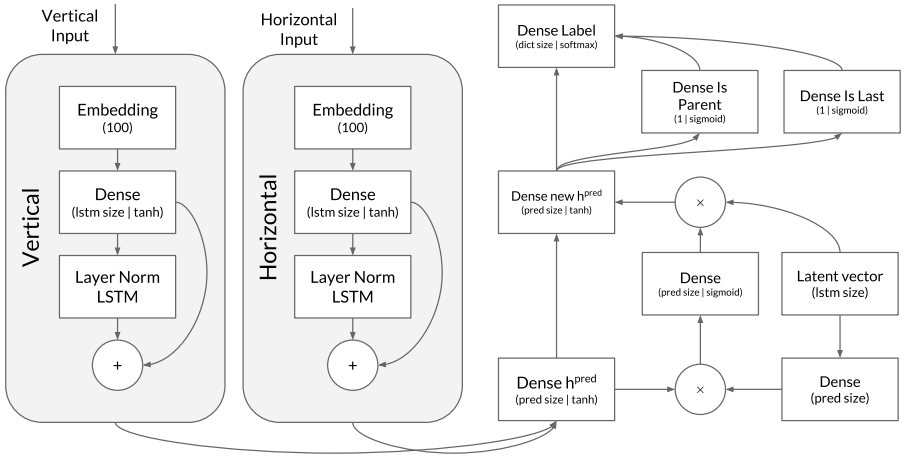

Figure 2: The modified DRNN cell consists of two LSTM blocks, each comprising embedding, input projection layer to adapt the input size to LSTM size, layer normalization LSTM cell, and residual skip connection between the projection and the output layers. DRNN cell uses the latent vector to enrich an prediction hidden state by the lost information during hierarchical processing.

in Fig. 3. Preliminary experiments with Python ASTs we conducted in (Anonymous, 1999) have shown that this brings significant improvements. We hypothesize that ASTs capture only program *syntax*, while pieces of code can be *semantically* related even if they do not reside in a common part of an AST. For instance, a reference to a previously used variable may require 'climbing up' the tree and then descending down to unrelated (non-sibling) subtrees.

We start by initializing both states of the vertical and the horizontal LSTM cells h_i^a, h_i^f with $h^{(latent)}(x)$. Alvarez-Melis & Jaakkola (2017) suggest to intialize only vertical hidden state with latent vector, however, our modification concerning the transfer of horizontal states in breadth-first order implies the same initialization approach also to the horizontal state. The DRNN merges both states of LSTMs in a *prediction state*:

$$h_i^{(pred)} = \tanh(U^f h_i^f + U^a h_i^a), \tag{8}$$

where U^f and U^a are learnable parameters. $h_i^{(pred)}$ is then mapped to the probability distribution of node's label

$$l_i = \text{softmax}(W h_i^{(pred)} + \alpha_i v^a + \varphi_i v^f), \tag{9}$$

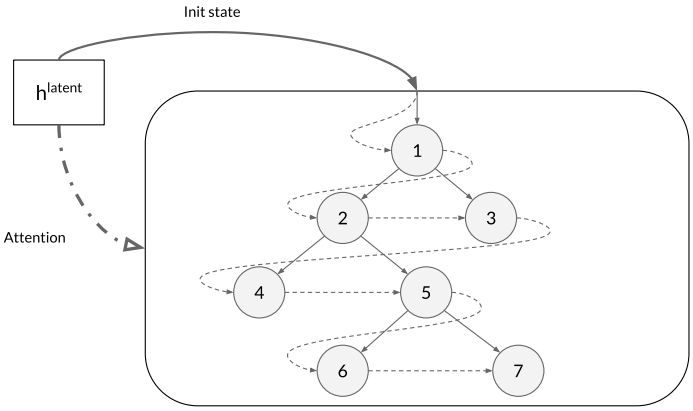

Figure 3: Propagation of the hidden states and latent vector during decoding. The dashed line illustrates the traversal of the horizontal state, the solid line propagation of the vertical state. The latent vector is used to initialize both states, but has also impact on subsequent decoding steps via an attention mechanism.

where $|l_i| = |v_i|$ (see the description of enc), and v^a and v^f are learnable parameters applied to *is-parent* flag (α_i) and *is-last-child* flag (φ_i). In training, these are used for *teacher forcing*, i.e. replaced with ground truth data according to the shape of the input tree x; in inference mode, the flags are calculated as:

$$\alpha_i^{(pred)} = \sigma(u^a h_i^{(pred)}) \tag{10}$$

$$\varphi_i^{(pred)} = \sigma(u^f h_i^{(pred)}) \tag{11}$$

where u^a and u^f are learnable parameters.

In training, we minimize the sum of *label loss* and *topological loss*:

$$L(x) = \sum_i L^{xent}(l_i, V(x_i)) + L^{bxent}(\alpha_i^{(pred)}, \alpha_i) + L^{bxent}(\varphi_i^{(pred)}, \varphi_i) \tag{12}$$

where $L^{xent}(l_i, V(x_i))$ is the categorical cross-entropy loss between the ground truth label $V(x_i)$ and the label l_i produced by the DRNN, and L^{bxent} is the binary crossentropy, applied separately to the *is-parent* flag and *is-last-child* flag.

Because successive iterations of the DRNN may arbitralily modify its prediction state $h_i^{(pred)}$, the information from the encoder may fade away with time, making it harder to reproduce the input tree x. A range of works, most relevantly Chen et al. (2018), have shown that this loss of information can be supplemented with an *attention mechanism*. However, the specific attention architecture proposed there assumes that the window of attention scans the corresponding nodes of the input tree x, which implies that x has to be available also during decoding. As we cannot assume this in our setting, where the decoder receives only the latent vector h^{latent} as input (Fig. 1), we come up with an alternative attention mechanism that relies only on h^{latent}. We define a (soft) attention window as

$$A = \sigma((h^{(latent)} U^{(A)} \odot h_i^{(pred)}) W^{(A)}) \tag{13}$$

and then use it to update the prediction state as follows (compare to the attention-less update formula (8)):

$$h_i^{(pred)} = \tanh((A \odot h^{(latent)}) C^{(A)} + h_i^{(pred)} C^{(pred)}) \tag{14}$$

where $U^{(A)}$, $W^{(A)}$, $C^{(A)}$ and $C^{(pred)}$ are learnable parameters. We expect this mechanism to learn focusing on the parts of $h^{(latent)}$ that carry the information that is relevant for reproduction of tree structure, particularly at the deeper levels.

2.3 Sentence-to-latent mapping

To map the sequences of word embeddings of the NL specification (300-dimensional vectors $(q_1, \ldots, q_n)$) to the latent space $h^{(latent)}$ (Fig. 1), we employ a multilayer bidirectional LSTM (Schuster & Paliwal, 1997; Graves et al., 2005). The dimensionality of LSTM's state and output is the same as that of $h^{(latent)}$. The output of the LSTM is passed through a $\tanh$ layer, in order to match the $(-1, 1)$ range of values produced by the tree encoder – which also uses $\tanh$ (Eq. 7) – and fed as $h^{(latent)}$ to the decoder.

Internally, this recurrent network is a 4-layer LSTM: there are four forward cells stacked upon each other, i.e. each consecutive cell receives the output of the previous cell as input, and analogously a stack of four backward cells. The final state of the topmost forward cell (reached after the forward pass over the input sequence) and the final state of the topmost backward cell (reached after the backward pass over the input sequence) are concatenated to form the final output.

3 Program Synthesis Tasks

We assess the performance of SAPS on the set of NL-based program synthesis tasks proposed in (Polosukhin & Skidanov, 2018). The domain-specific language (DSL) used in this suite is Lisp-based AlgoLisp, defined by the grammar in Fig. 4, reproduced from (Polosukhin & Skidanov, 2018). A single program is composed as a nested list of instructions and operands of three types: `string`, `Boolean` and `function`. The language features a number of standard functions (cf. the last-but-one production in the grammar). The motivation for choosing this dataset was the large number

```
program       ::= symbol
symbol        ::= constant | argument | function_call | function | lambda
constant      ::= number | string | True | False
function_call ::= (function_name arguments)
function      ::= function_name
arguments     ::= symbol | arguments , symbol
function_name ::= Reduce | Filter | Hap | Head | + | - | ...
lambda        ::= lambda function_call
```

Figure 4: The grammar of the AlgoLisp DSL from (Polosukhin & Skidanov, 2018).

of examples and brevity of NL specifications (in contrast to the programming contest data we used in (Anonymous, 1999)). Moreover, the simplicity of syntax causes the AlgoLisp programs to be essentially equivalent to their AST trees. See (Polosukhin & Skidanov, 2018) for detailed description of the dataset.

The dataset comprises 99506 examples, each being a pair of NL specification and the corresponding AlgoLisp program. An example can be seen in Table 1. To handle the variables that occur in programs (like a in the above example), we use the same approach as Polosukhin & Skidanov (2018), i.e., each occurrence of a new variable in the program allocates a new *placeholder*, and the placeholders extent the vocabulary of tokens used in the NL embedding. Placeholders are common for all examples, i.e. the first occurrences of variables in all programs are mapped to the same, first placeholder. The descriptions in NL does not specify the variables, which are needed in source code, therefore there were no need to resorting to such placeholders on the encoder side.

For conformance with Polosukhin & Skidanov (2018), we split the data into training set of 79214 examples, validation set (9352 examples) and test set (10940 examples). The lengths of average NL specification and associated code are, respectively, 38 and 24 tokens. The shortest and longest description contain 8 and 164 tokens respectively (2 and 217 for code snippets).

It may be worth noting that, in contrast to the relatively large number of examples available, the vocabularies of terms are rather humble: there are only 281 unique terms/tokens in the NL specifications, and only 72 unique terms in the AlgoLisp programs. The NL vocabulary used in specifications is strongly related to programming, containing words rarely used in everyday language, lie *prime, inclusive, incremented* etc.

After closer examination, we **identified potential inconsistency in the dataset**: for a about 10% of tasks (in the test set), some tests are not passed even by the ground truth program (we verified that using the AlgoLisp interpreted published in (Polosukhin & Skidanov, 2018)). To address this issue, we decided to use two performance measures: one using the unmodified test and validation sets, in order to maintain comparability to results presented in (Polosukhin & Skidanov, 2018), and another one using a filtered test set, containing only the tasks that were free from the above problem. Note however that this issue does not affect the training of SAPS, as it does not involve tests.

4 EXPERIMENT

The method has been implemented in TensorFlow Abadi et al. (2016). We train the autoencoder using Adam optimizer (Kingma & Ba, 2014) with initial learning rate set to initial value of 5×10^{-4}, and then reduced exponentially at rate 0.975 per every two epochs. To lessen the risk of overfitting, we applied L2 regularization to each of trainable parameter, except for biases and layer

Table 1: An example AlgoLisp program (right) and its NL specification (left).

You are given an array a. Find the smallest element in a, which is strictly greater than the minimum element in a	`(reduce (filter a (partial0 (reduce a inf) <)) inf min)`

Table 2: Direct comparison with results achieved by Polosukhin & Skidanov (2018). Accuracy is measured as the percentage of programs passing all tests.

Model	Dev Acc	Test Acc
Attentional Seq2Seq	54.4%	54.1%
Seq2Tree	61.2%	61.0%
SAPS (NLP2Tree - 256)	**85.29%**	**83.03%**
Seq2Tree + Search (only for comparison)	86.1%	85.8%

normalization. When applied to the validation set, the autoencoder reproduces successfully 72.98% of AST trees in the best configuration.

Given the trained decoder and GloVe embedding (Section 2.1), we train the sentence-to-tree and sentence-to-vector mapping using the same configuration and methods as those used for training the autoencoder.

4.1 RESULTS

We examine network's ability to fully reconstruct trees, the number of generated programs that pass all tests (and at least 50% of tests), and confront SAPS with other methods of synthesis of programs from NL. In addition, we conducted an analysis of the sensitivity of the network to changes in input data (natural language). None of the considered trained configurations of SAPS produced any syntactically incorrect programs (average error 0.0%). The generated programs by the best obtained network for examples from validation set passed in average 94.35% of unit tests, while from the test set 93.32%.

Due to the inconsistencies in AlgoLips dataset described in Section 4.1, we divided the results into two groups. First, we compare our best configuration, SAPS with the latent layer of size 256, directly to those achieved by (Polosukhin & Skidanov, 2018), using their evaluation code. The results are presented in Table 2. Though the best test-set accuracy reported by Polosukhin & Skidanov (2018) is 85.8%, achieving it requires a sophisticated search mechanism and use of additional tests. On the other hand, our approach is entirely neural, so it seems fair to compare it to analog neural-only architectures from that cited study. The best accuracy obtained by Polosukhin & Skidanov (2018) without search is 61.0% on the test set. It should be noted that, despite involving no explicit search mechanism, our fully neural algorithm is only slightly worse than the best search-based approach from (Polosukhin & Skidanov, 2018). Moreover, SAPS outperforms the general attention mechanism used in *Seq2Tree* by roughly 24%.

Secondly, we evaluated the performance of different configurations of our model. In Table 3, we present the accuracy of several variants of SAPS, measured as the percentage of perfectly synthesized programs (i.e., such that the generated AST trees is identical to the target one), separately for the validation (developmental) set and test set. We report two performance metrics: the accuracy of autoencoder (tree2tree), i.e. the percentage of perfectly recreated AST trees in that training phase, and the overall accuracy of SAPS. We also experimented with different sizes of the latent layer: 64, 128, and 256. As expected, the configurations involving largest hidden states perform the best. Also, each complete SAPS architecture is better at synthesizing programs from NL specifications than the

Table 3: Percentages of synthesized programs that are syntactically identical to the target ones (same AST tree).

Model	Validation	Test	Model	Validation	Test
tree2tree - 256	72.98%	68.42%	**SAPS - 256**	**92.68%**	**91.18%**
tree2tree - 128	67.06%	60.18%	SAPS - 128	82.59%	60.18%
tree2tree - 64	50.24%	39.36%	SAPS - 64	58.65%	52.22%
			NLP2Vec - 256	14.91%	14.39%

Table 4: Percentage of programs passing all tests, using all examples provided in the AlgoLisp database (Polosukhin & Skidanov, 2018) (*All*), and using only the tests that are consistent with the target program (*Only passable*). See the main text for detailed explanation.

Model	All tests				Only passable tests			
	Validation		Test		Validation		Test	
	Accuracy	50-Accuracy	Accuracy	50-Accuracy	Accuracy	50-Accuracy	Accuracy	50-Accuracy
SAPS - 256	85.29%	88.34%	83.03%	86.58%	93.24%	94.27%	91.97%	93.22%
SAPS - 128	76.62%	80.92%	70.91%	75.89%	83.90%	86.41%	78.56%	81.54%
SAPS - 64	55.08%	60.35%	49.80%	56.31%	60.19%	64.31%	55.17%	60.56%

corresponding constituent autoencoder is at reproducing the AST trees. We hypothesize that this is due to relatively simplicity of the TreeLSTM encoder used here, which features single, feedforward LSTM, contrary to multilayer, bidirectional LSTM used in the NL encoder.

In the configurations discussed above, the decoder (taken from the trained autoencoder) learns alongside with the NL-to-latent mapping. For comparison, we consider also a variant dubbed NLP2Vec. In this setting, we used the latent vectors from pretrained autoencoder (tree2tree) model as the target (desired output) for the NL-to-latent mapping, and trained the network to minimize the distance between the predicted vector and the one computed by tree2tree (while the decoder remained fixed). As expected, the obtained results were inferior to others (outperforming, however, the approach based only on input-output pairs from (Polosukhin & Skidanov, 2018), which achieved accuracy of 13.3% in the best case (with search mechanism)).

In Table 4, we compared the results obtained on the original, unmodified version of AlgoLisp dataset with the results obtained on tasks containing only valid, passable tests (see the note on this issue in Section). Expectedly, networks obtained higher scores when evaluated only on the tasks that are free from the abovementioned issue, and those scores are also higher than the percentages of programs that were syntactially identical to the target (Table 3). The latter fact should not be surprising, as a syntactically perfect program has to pass all tests.

To validate the obtained results, we evaluated the robustness of SAPS by applying a range of modifications to the NL specification (Table 5). We checked how does the generated source code depend on different levels of input complexity (*simplification*). We provided a set of NL descriptions and sorted them from the longest to the shortest. To our surprise, the network works well even if the input is very laconic. We evaluated also the capabilities of generalization in response to various replacements in input sequences, like for example replacing operations (*multiplication* $\rightarrow$ *minimum*) and applying different ranges of arrays. We also tested the robustness to complex replacements consisting of multiple simpler ones.

5 Related work

With the recent advent of powerful hardware, the problem of program synthesis has gained a large attention from computer scientists. The number of possible applications is virtually infinite, however a few main approaches to this problem could be identified.

Code synthesis from input / output examples
Recently a handful of methods utilizing input / output examples for program synthesis were introduced. Balog et al. (2016) used a neural network as a guide for a search algorithm. The network predicts the most probable tokens, which greatly reduces the number of solutions that need to be evaluated during search. Krawiec et al. (2017) use genetic programming to find a set of programs that match the formal specification, which ensures the correctness of generated programs for all possible inputs from specification. Devlin et al. (2017) create programs using noisy input / output pairs.

Differentiable compilers
Another approach is to avoid using a learning algorithm to create a human readable code, but to embed the whole proces implicitly in the algorithm. These methods often utilize neural networks and could be seen as differentiable compilers. Graves et al. (2014) simulate a Turing machine

using neural network, endowing it with differentiable memory. A similar approach was proposed by Sukhbaatar et al. (2015), where a memory was also used.

Source code and AST trees from natural language description
With the recent advances in natural language understanding, a number of approaches utilizing natural language descriptions for code synthesis were introduced. Rabinovich et al. (2017); Yin & Neubig (2017) generate code in the form of abstract syntax trees instead of plain source code. They use recursive neural decoders, specialized for the abstract syntax tree generation. An interesting approach is presented by Ling et al. (2016), where the authors use a set of neural modules to generate source code from multimodal inputs (text and quantitative features), based on trading card game.

A neural guided approach was proposed by Polosukhin & Skidanov (2018). The authors used a neural network to guide a search algorithm. They used a sequential encoder to fold a short description in natural language and then a tree decoder to generate a set of most possible nodes in the AST. For each node, the proposed labels were evaluated by search algorithm on a set of input / output tests. Then, the proces was repeated for next node, recurrently. The authors used the attention mechanism (Bahdanau et al., 2014) on the whole input sequence, contrary to our approach, where attention is based only on latent vector.

6 Discussion

SAPS manages to achieve state-of-the-art test-set accuracy, on par with that of Polosukhin & Skidanov (2018), and does so with a bare neural model, without any additional postprocessing or other forms of guidance. This remains in stark contrast to that study, where network was queried re-

Table 5: The effects of various types of modifications of the NL specification. The first specification in each group is an original task from the validation set, and those that follow are its modified variants (with most modifications marked in bold). In *Simplification*, we drop and swap words from the specification until the output program is not correct anymore. *Simple replacement*: we aim at modifying a single keyword in the program. *Complex replacement*: change of entire subexpression in the output program. *Makeover*: Complex modification of specification. Except for the specification *a b, a + b*, all synthesized programs are consistent with specification.

Specification	Synthesized program
Simplification	
you are given numbers a and b, your task is to find a + b	`(+, a, b)`
you given numbers a b, your is find a + b	`(+, a, b)`
*given **a numbers b**, find a + b*	`(+, a, b)`
given a numbers b, a + b	`(+, a, b)`
a b, a + b	`(+, (+, a, b), c)`
Simple replacement	
*you are given numbers a and b, your task is to find **a multiplied by b***	`(*, a, b)`
*you are given numbers a and b, your task is to find **minimum a and b***	`(min, a, b)`
Complex replacement	
*given a number a and an array of numbers b, find the length of the longest subsequence of range from **0 to a inclusive** that is a prefix of b*	`(reduce, (range, 0, (+, a, 1)), 0, (lambda2, (if, (==, arg2, (if, (<, arg1, (len, b)), (deref, b, arg1), 0)), (+, arg1, 1), arg1)))`
*given a number a and an array of numbers b, find the length of the longest subsequence of range from **1 to a exclusive** that is a prefix of b*	`(reduce, (range, 1, a), 0, (lambda2, (if, (==, arg2, (if, (<, arg1, (len, b)), (deref, b, arg1), 0)), (+, arg1, 1), arg1)))`
Makeover	
given an array of numbers a, find median of values in a after only keeping first half	`(deref, (sort, (slice, a, 0, (/, (len, a), 2))), (/, (len, (slice, a, 0, (/, (len, a), 2))), 2))`
given an array of numbers a, find mean of values in a after only keeping second half	`(/, (reduce, (slice, a, (/, (len, a), 2), (len, a)), 0, +), (len, (slice, a, (/, (len, a), 2), (len, a))))`

peatedly in a Tree-Beam search heuristics to produce the target program step by step, testing the candidate programs on provided tests (see Fig. 2 in (Polosukhin & Skidanov, 2018)), and in contrast to Balog et al. (2016), where a network was used to prioritize search conducted by an external algorithm. SAPS's architecture can provide a similar level of quality with purely neural mechanisms. This seems interesting in itself; in particular, it is worth realizing that the fixed-dimensionality latent layer $h^{(latent)}$ implements an embedding for a large number of programs, many of them implementing sensible semantics - those present in our training, validation and testing set, but possibly also other programs (as suggested by the effects of manipulations shown in Table 5).

We attribute the high performance of SAPS mainly to the quite sophisticated design of our decoder, and in particular its susceptibility to learning. This claim is supported by the comparison of NLP2Tree to tree2tree (Table 3), where the latter used a fixed decoder, trained only in the autoassociative mode (Section 2.2). The gap between the performances of these architectures clearly indicates that it was essential to allow the decoder to adapt in the end-to-end learning spanning from NL to AST trees. This gives rise to an interesting question for follow-up studies: how much of autoassociative learning is required for SAPS to perform well? In an extreme scenario, it would be interesting to examine the performance of SAPS without any autoassociative pretraining at all.

Our networks produce exactly the required program in over 90% of cases (see Table 3). Even when the synthesized program is not identical to the target one, there is still a chance for it to be correct. For instance, for the test set, SAPS-256 produces a perfect copy of the target program in 91.18% of cases (Table 4), while 91.97% programs pass all tests (Table 3). Therefore, 0.79% of programs (roughly one in 11 of imperfectly reproduced programs) pass all tests despite being syntactically different from the target program. It seems likely that in those cases the network came up with an alternative implementation of the target concept expressed by the NL specification. One should keep in mind, however, that passing the tests does not prove semantic equivalence. This could be achieved with formal verification, which we leave out for future work.

In relation to that, let us also note that a substantial fraction of programs that do not pass all tests, pass at least 50% of them (see the 50-Accuracy in Table 4). Because the a priori probability of passing a test by a program is miniscule, this suggests that even the imperfect programs feature useful code snippets that make them respond correctly to some tests. All these promising values of performance indicators make SAPS's applications in real-world setting quite realistic.

It would be naive to expect SAPS to scale well for even longer specifications and/or target programs (the median length of the former was 36 words, and of the latter 20 tokens). That would arguably require augmenting the method with additional modules, for instance an attention mechanism for interpretation of the NL specification, as used by Polosukhin & Skidanov (2018). That is particularly true for tasks that involve complex, multi-sentence NL specifications.

7 Conclusion

We have shown that end-to-end synthesis of nontrivial programs from natural language is possible with purely neural means, and using exclusively gradient descent as the learning mechanism. Given that (i) capturing the semantics of the NL is in general difficult, (ii) there are deep differences between NL and AST trees on both syntactic and semantic level, and (iii) the correspondence between the parts of the former and the latter is far from trivial, we find this result yet another promising demonstration of the power dwelling in the neural paradigm. In future works, we plan to perform ablation studies on SAPS to determine which components are critical for its performance, and focus on the compositionality of both specifications and program code.

Acknowledgment. We thank the authors of (Polosukhin & Skidanov, 2018) for publishing their code and data.

References

Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Imple-*

mentation (OSDI 16), pp. 265–283, 2016. URL https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

David Alvarez-Melis and Tommi S. Jaakkola. T ree-structured decoding with doubly-recurrent neural networks. 2017.

Anonymous. *[To be disclosed after acceptance.]*. PhD thesis, 1999.

Jimmy Ba, Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. *CoRR*, abs/1607.06450, 2016.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.

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

Xinyun Chen, Chang Liu, and Dawn Xiaodong Song. Tree-to-tree neural networks for program translation. *CoRR*, abs/1802.03691, 2018.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy I/O. In *ICML*, volume 70 of *Proceedings of Machine Learning Research*, pp. 990–998. PMLR, 2017.

Alex Graves, Santiago Fernández, and Jürgen Schmidhuber. Bidirectional lstm networks for improved phoneme classification and recognition. In *Proceedings of the 15th International Conference on Artificial Neural Networks: Formal Models and Their Applications - Volume Part II*, ICANN'05, pp. 799–804, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-28755-8, 978-3-540-28755-1. URL http://dl.acm.org/citation.cfm?id=1986079.1986220.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines, 2014. URL http://arxiv.org/abs/1410.5401. cite arxiv:1410.5401.

Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *CoRR*, abs/1412.6980, 2014. URL http://dblp.uni-trier.de/db/journals/corr/corr1412.html#KingmaB14.

Krzysztof Krawiec, Iwo Błądek, and Jerry Swan. Counterexample-driven genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, pp. 953–960, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4920-8. doi: 10.1145/3071178.3071224. URL http://doi.acm.org/10.1145/3071178.3071224.

Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, Fumin Wang, and Andrew Senior. Latent predictor networks for code generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 599–609. Association for Computational Linguistics, 2016. doi: 10.18653/v1/P16-1057. URL http://www.aclweb.org/anthology/P16-1057.

Tomas Mikolov, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1532–1543. Association for Computational Linguistics, 2014. doi: 10.3115/v1/D14-1162. URL http://aclweb.org/anthology/D14-1162.

Illia Polosukhin and Alex Skidanov. Neural program search: Solving programming tasks from description and examples. pp. 11, 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)*, pp. 1139–1149. Association for Computational Linguistics, 2017. doi: 10.18653/v1/P17-1105. URL http://www.aclweb.org/anthology/P17-1105.

Mike Schuster and Kuldip K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45:2673–2681, November 1997.

Sainbayar Sukhbaatar, arthur szlam, Jason Weston, and Rob Fergus. End-to-end memory networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 28*, pp. 2440–2448. Curran Associates, Inc., 2015. URL http://papers.nips.cc/paper/5846-end-to-end-memory-networks.pdf.

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

Pengcheng Yin and Graham Neubig. 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)*, pp. 440–450. Association for Computational Linguistics, 2017. doi: 10.18653/v1/P17-1041. URL http://www.aclweb.org/anthology/P17-1041.