Tree Variational Autoencoder for Code *

Vadim Liventsev[1,2][0000−0002−6670−6909], Sander de Bruin[1] Aki Härmä[2], and Milan Petković[1,2][0000−0002−6264−6822]

[1] Eindhoven University of Technology
Eindhoven, the Netherlands
{v.liventsev,m.petkovic}@tue.nl
[2] Philips Research
High Tech Campus 34, 5656 AE Eindhoven, the Netherlands
{vadim.liventsev,milan.petkovic}@philips.com

Abstract. Autoencoder models of source code are an emerging alternative to autoregressive large language models with important benefits for genetic improvement of software. We hypothesise that encoder-decoder architectures that are currently standard in both natural and machine language processing are suboptimal for source code because they ignore the grammatical structure of programs that can be trivially and robustly derived with an Abstract Syntax Tree, AST, parser. We propose a structured Variational Auto-Encoder, VAE, based on Child-Sum TreeLSTM that operates directly on the AST of the program. We train it, along with a baseline sequence VAE, on a dataset of competitive programming submissions and find that the structured model demonstrates better performance in most tests, with some notable exceptions.[3]

Keywords: deep learning, machine learning on source code, autoencoder models, foundation models

1 Introduction

1.1 Foundation models for code

The paradigm of foundation models [1] has recently been very prominent in machine learning and machine learning on source code is no exception. In this paradigm, a model is first trained on a large dataset to solve a generic task such as next-token prediction and then used as a central component in solutions of various specific tasks. The foundation models for source code, based on architectures such as GPT Codex [2–5] and BERT [6, 7] have enabled significant process in tasks like programming by example [8,9] and even human-comparable performance in coding competitions [10].

* This work was paritally funded by the European Union's Horizon 2020 research and innovation programme under grant agreement n° 812882. This work is part of "Personal Health Interfaces Leveraging HUman-MAchine Natural interactionS" (PhilHumans) project

[3] See `tree2tree.app` for a demo. The model is downloadable at `https://github.com/sander102907/autoencoder_program_synthesis`.

1.2 Autoencoder genetic programming

While undoubtedly useful in many program synthesis tasks, popular foundation models may fall short in the areas of genetic programming [11] and genetic improvement of software [12]. In this settings, new programs are found by exploring the space of programs similar to one or several reference programs. The task of applying these perturbations to programs could benefit from a foundational model, however, it's unclear how to achieve this with the current autoregressive models. Autoencoder Genetic Programming [13–15] argues for using autoencoder [16] models instead. These models embed programs into a high-dimensional vector space, making it easy to mutate a program by add random noise to the embedding vector or combine several programs by averaging their embedding vectors.

1.3 Structural language models of code

Unlike natural languages, programming languages are easier to represent structurally due to the nature of their syntax which improves machine learning performance. Prior work on structural code encoding [17,18], structural code decoding [19, 20] and structural encoder-decoder models for machine translation [21] suggests that models that operate directly on the tree structure of the program can achieve better performance than models that operate on a sequence of tokens.

To the best of our knowledge, however, the advantage of structural models has not been tested in autoencoders for genetic programming. [22, 23] find that it is beneficial to include grammatical metadata in token representations for a traditional sequence model, but do not employ a tree-based encoder-decoder architechure.

Thus the research question we set out to answer is: *would an model that operates directly on the program's Abstract Syntax Tree learn a better latent representation of the source code than a model that operates on a sequence of tokens?*

2 Proposed architecture

2.1 Autoencoder type

We have chosen to use a variational autoencoder [24] since, unlike its non-stochastic counterpart,it is less dependent on choosing the right size of the latent vector, since the Kullback Leibner component encourages the model to use as small of a subspace of the latent space as it can. Our experiments do, however, indicate that the choice of latent dimension is still important.

2.2 Encoder

The encoder network aims to capture the most relevant information in a program and map it to a smaller representation.

Embedding layer The first layer of the encoder network convert tokens into dense representations, which can be either initialized randomly or initialized with pre-trained parameters and then fine-tuned further.

Tree-LSTM We employ the Child-Sum Tree-LSTM [25] which is defined as follows. Given some tree, we can denote the set of children of a node y as $C(y)$ and the vector representation of the node as $\mathbf{x}^y$. The transition equations between the different Tree-LSTM are the following:

$$\mathbf{h}^y_* = \sum_{z \in C(y)} \mathbf{h}^z \tag{1}$$

$$\mathbf{i}^y = \sigma(\mathbf{W}_i \cdot \mathbf{x}^y + \mathbf{U}_i \cdot \mathbf{h}^y_* + \mathbf{b}_i) \tag{2}$$

$$\mathbf{f}^{yz} = \sigma(\mathbf{W}_f \cdot \mathbf{x}^y + \mathbf{U}_f \cdot \mathbf{h}^z + \mathbf{b}_f) \tag{3}$$

$$\mathbf{o}^y = \sigma(\mathbf{W}_o \cdot \mathbf{x}^y + \mathbf{U}_o \cdot \mathbf{h}^y_* + \mathbf{b}_o) \tag{4}$$

$$\mathbf{u}^y = tanh(\mathbf{W}_u \cdot \mathbf{x}^y + \mathbf{U}_u \cdot \mathbf{h}^y_* + \mathbf{b}_u) \tag{5}$$

$$\mathbf{c}^y = \mathbf{i}^y \odot \mathbf{u}^y + \sum_{z \in C(y)} \mathbf{f}^{yz} \odot \mathbf{c}^z \tag{6}$$

$$\mathbf{h}^y = \mathbf{o}^y \odot tanh(\mathbf{c}^y) \tag{7}$$

In eq. (4), $z \in C(y)$ and $\odot$ denotes the element-wise product (Hadamard product), whereas σ and tanh refer to elementwise sigmoid and hyperbolic tangent. W, U and b denote trainable parameters of the model. Note that node y depends on the hidden states of all of the children $C(y)$, in other words, the computation order of the Tree-LSTM is bottom-up. Just like with the standard LSTM model, Tree-LSTM can be stacked to create a multilayer Tree-LSTM. In such a multilayer architecture, the hidden state of a Tree-LSTM unit in layer l is then used as input to the Tree-LSTM unit in layer $l + 1$ in the same time step, the same as with the standard LSTM [26]. The idea is to let higher layers capture longer-term dependencies of the input. In the case of Tree-LSTMs, this translates to capturing longer-term dependencies along the paths of a tree.

Neural attention Tree-LSTM layers are followed by an attention layer. The node importance calculation is based on [27], and updates the hidden states as follows:

$$\mathbf{h}_{at} = \mathbf{h} \cdot tanh(\mathbf{W} \cdot \mathbf{h} + \mathbf{b}) \tag{8}$$

Here, h denotes the hidden states of the last Tree-LSTM layer. This additional layer allows the network to prioritize nodes that contain the most information.

Pooling The last step in the architecture is the pooling layer responsible for compressing the sequence of h_{at} into a fixed size vector. We rejected a common [28] approach of only taking the last hidden state of the RNN as the input to the decoder due to the long-term memory loss problem [29] and use max pooling instead, see figure fig. 1.

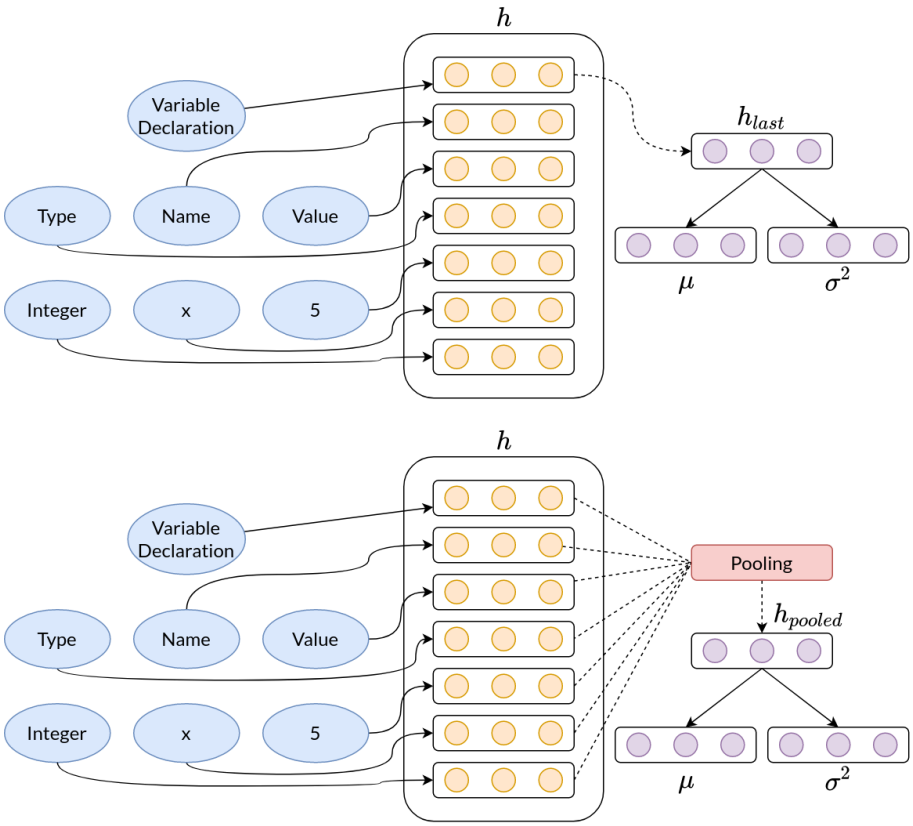

Fig. 1: **Top**: Typical architecture of encoder model of VAE in which only the last hidden state from the RNN is used to compute the mean u and variance σ^2. **Bottom**: A pooling method to aggregate the hidden states from the RNN to compute the mean u and variance σ^2.

Sampling latent code The pooled vector is then used to compute the mean and variance of the approximate posterior to sample a latent code z with the help of the reparametrization trick [24]. The mean and variance are computed using linear layers that learn a set of weights and biases.

2.3 Decoder

The goal of the decoder network is to reconstruct a given input as accurately as possible, given the latent code produced by the encoder.

Tree decoding We use the same tree structure for decoding as we used for encoding. Having a reversed order of the input sequence compared to the reconstructed sequence has been shown in [28] to improve the performance of the model. We

employ this technique in our model, which means that since our encoder processes trees bottom-up, the decoder will produce trees top-down. The idea here is that the first steps of decoding the tree are more related to the latent space than the last steps.

A method called the doubly-recurrent neural network (DRNN) [30] allows for top-down tree generation from an encoded vector representation. This method operates solely on the vector representation and does not require that either the tree structure or the nodes are given. The DRNN is based on two recurrent neural networks, breadth and depth-wise, to model ancestral and fraternal information flow. For some node y with parent $pa(y)$ and previous sibling $s(y)$, the ancestral and fraternal hidden states are computed as follows:

$$\mathbf{h}_a^y = rnn_a(\mathbf{h}_a^{pa(y)}, \mathbf{i}^{pa(y)}) \tag{9}$$

$$\mathbf{h}_f^y = rnn_f(\mathbf{h}_f^{s(y)}, \mathbf{i}^{s(y)}) \tag{10}$$

Where rnn_a, rnn_f are functions that apply one step of the ancestral and fraternal RNNs, respectively. Furthermore, $\mathbf{i}^{pa(y)}$, $\mathbf{i}^{s(y)}$ are the input values (label vectors) of the parent and previous sibling respectively. After the ancestral and fraternal states of y have been computed with the observed labels of its parent and previous sibling, these states can be combined to form a predictive hidden state:

$$\mathbf{h}_{pred}^y = \tanh\Big((\mathbf{W}_a \cdot \mathbf{h}_a^y + \mathbf{b}_a) + (\mathbf{W}_f \cdot \mathbf{h}_f^y + \mathbf{b}_f)\Big) \tag{11}$$

Where the operations applied to $\mathbf{h}_a^y$, $\mathbf{h}_f^y$ are linear layers with learnable weights and biases. This combined state then contains information about the nodes' surroundings in the tree.

For each node in the tree, the model needs to decide whether it has offspring and whether it has any successor siblings. We can use the predictive hidden state of a node $\mathbf{h}_{pred}^y$, with a linear layer and a sigmoid activation to compute the probability for offspring and successor siblings as:

$$p_a^y = \sigma(\mathbf{W}_{pa} \cdot \mathbf{h}_{pred}^y + \mathbf{b}_{pa}) \tag{12}$$

$$p_f^y = \sigma(\mathbf{W}_{pf} \cdot \mathbf{h}_{pred}^y + \mathbf{b}_{pf}) \tag{13}$$

During training, we use the actual values for whether a node has children and successor siblings. During inference, we can either greedily choose any confidence level to continue creating offspring and succeeding siblings by checking whether the probability is above some threshold or sample this choice.

Besides topological predictions, the model should also predict the label of each token. Again the predictive hidden state can be used for label prediction as follows:

$$\mathbf{o}^y = softmax\Big(\mathbf{W}_o \cdot \mathbf{h}_{pred}^y + \mathbf{b}_o\Big) \tag{14}$$

Tree decoding optimizations Now that we have the basic DRNN model [30] in place to generate a tree from scratch using a latent vector, we can optimize it for our use case.

The first issue is the possibly infinitely large vocabulary that source code allows. Since progam behavior is invariant to identifier replacement we map each unique identifier to a reusable ID [31] and treat the prediction of identifiers as a clustering problem between declarations and references. We use the predictive hidden states of the nodes to learn relationships between declarations and references.

The model can keep track of a list of the declared identifiers while generating an AST. Each time a new identifier is declared, a new reusable ID is added to the list. Then for each reference, we can compute the similarity to each of the declared identifiers using some similarity function and predict the most similar identifier. We can keep track of what type of node we are currently trying to predict due to the AST structure and because we have access to the parent node label, i.e., the parent node indicates whether the child node is a declaration or reference. Let D be the set of currently declared identifier nodes and y be the current reference node we are trying to predict, the most similar declared identifier can be computed as follows:

$$\mathbf{s}^{yz} = similarity(\mathbf{W}_c \cdot \mathbf{h}^y_{pred} + \mathbf{b}_c, \mathbf{W}_c \cdot \mathbf{h}^z_{pred} + \mathbf{b}_c) \tag{15}$$

$$\mathbf{r}^y = \min_{z \in D}(\mathbf{s}^{yz}) \tag{16}$$

We have a similar problem for literal tokens; developers can use an almost infinitely large number of unique literals in source code. However, in contrast to identifier tokens, literal tokens influence the functionality of a program. Therefore, to assure that generated programs are still compile-able, we cannot remap the literal tokens to reduce the token count. For example, we cannot map rarely used literals to special unknown tokens, as unknown tokens create compiler errors. Instead, we can employ adaptive softmax [32] to use a vocabulary consisting of many unique literal tokens without a considerable increase in computational complexity.

We have identifiers and literals as token categories already, but we can also categorize the leftover tokens into the following categories:

- **Reserved tokens:** for, if, while, ...
- **Types:** int, long, string, ...
- **Built in function names:** printf, scanf, push, ...

In total, the five categories cover all the different tokens of the programming language (at least for C++). The reason for splitting up the leftover tokens into more categories is to predict these categories separately based on their parent node. For example, this ensures that we do not input a type-token in the tree, where there should be a reserved token. The categorization improves the compilation rate of the generated programs by allowing the model only to

predict tokens of the correct token category. The tree-structured representation during decoding allows us to use this optimization technique. For the reserved tokens, type, and built-in function names, eq. (14) is used for label prediction, as there is only a limited number of unique tokens in these categories.

To allow for the categorized label predictions, we need to add one more element to the DRNN model. An essential aspect of the tree structure is that identifiers, built-in function names, and literals occur in the leaves of the trees. Hence if a node has offspring, the category of the current node must be a reserved token. However, if a node has no offspring, it can be either of the categories, and we need to somehow decide which category to predict a label for. Note that a reserved token can also be on a leaf node on the tree. For example, consider an empty return statement. For that reason, similar to the topology predictions, we have the model predict whether a node is of the reserved token category or not. This prediction is computed in the same way as the topology predictions using the predictive hidden state of the node as follows:

$$p_r^y = \sigma(\mathbf{W}_{pr} \cdot \mathbf{h}_{pred}^y + \mathbf{b}_{pr}) \tag{17}$$

Add gate The DRNN model has a large flaw, where it is not able to differentiate between paths with the same prefix. For example, consider the situation depicted in fig. 2, where we have two function declarations named 'add' and 'main'. Due to the information flow downwards, both name nodes have the same hidden state and the model is not able to distinguish the leaf nodes and will therefore predict the same label for both. This issue is depicted in the left image of fig. 2. To solve this issue, we would like to incorporate the fraternal states in the downwards flow for the model to learn to differentiate the paths downwards. Hence we would like to revise eq. (10), where we take inspiration from the LSTM model and apply the idea of the add gate to our ancestral update formula as follows:

$$\mathbf{m}_f^y = \sigma(\mathbf{W}_m \cdot \mathbf{h}_f^y + \mathbf{b}_m) \tag{18}$$

$$\mathbf{a}_f^y = tanh(\mathbf{W}_a \cdot \mathbf{h}_f^y + \mathbf{b}_a) \tag{19}$$

$$\mathbf{h}_a^y = \mathbf{h}_a^y + (\mathbf{a}_f * \mathbf{m}_f) \tag{20}$$

This update to the fraternal state is applied after predicting the label for node y, which is depicted in the right image of fig. 2. Here, a_f^y is the value of the transformation on the previous sibling state that should be added to the parent state, where the *tanh* transforms it between -1 and +1 to mitigate exploding gradients. Furthermore, m_f^y decides which elements should be added using a sigmoid function that outputs values between 0 and 1. By multiplying a_f^y with m_f^y, the model can learn to decide what and how much to add from the previous sibling state to each parent state's element to help predict the next steps of the tree.

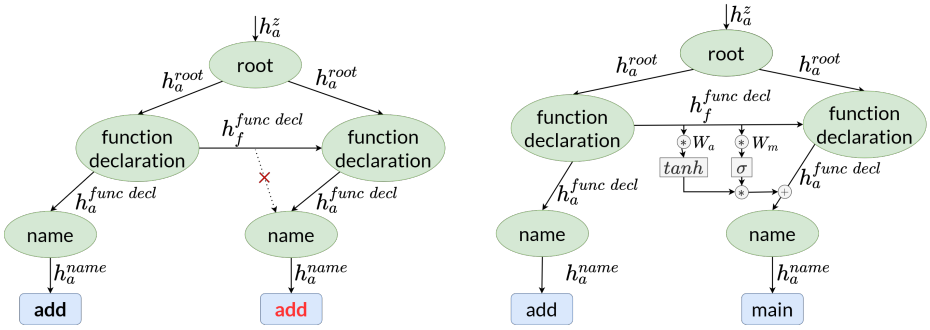

Fig. 2: DRNN expanded with an add gate to allow for information flow from previous siblings downwards

2.4 Optimization

Mitigating KL vanishing KL vanishing is a common issue when dealing with VAEs with a decoder parameterized by an auto-regressive model. We mitigate it vanishing using cyclical KL cost annealing [33]. Furthermore, we apply pooling to the hidden states of the RNN network in the encoder. Long *et al.* [34] show this pooling method can effectively prevent the posterior collapse issue.

Loss function Predicting whether a node has offspring and successor siblings are binary choices, so we can use binary cross-entropy to compute the loss for predicting the topology of the AST. Let a^y, f^y represent the actual values of having offspring and successor siblings for node y, the topological losses for this node are then computed as follows:

$$\mathcal{L}_a(y) = -a^y \cdot \log(p_a^y) + (1 - a^y) \cdot \log(1 - p_a^y) \tag{21}$$

$$\mathcal{L}_f(y) = -f^y \cdot \log(p_f^y) + (1 - f^y) \cdot \log(1 - p_f^y) \tag{22}$$

where $\mathcal{L}_a$, $\mathcal{L}_f$ denote the ancestral and fraternal loss respectively. Because the reserved token category prediction (eq. (17)) is so similar to the topological predictions, the loss for that component can be defined in a similar fashion:

$$\mathcal{L}_r(y) = -r^y \cdot \log(p_r^y) + (1 - r^y) \cdot \log(1 - p_r^y) \tag{23}$$

where we define r^y to represent the actual value of node y being in the reserved token category.

Label prediction is a classification problem for all label categories, except the identifiers. Hence, we can compute the cross entropy loss (or negative log likelihood):

$$\mathcal{L}_l(y) = -\log(\mathbf{o}^y[l^y]) \tag{24}$$

where we assume that l^y is the index of the true label, and hence $\mathbf{o}^y[l^y]$ retrieves the softmax value at the index of the correct class.

Lastly, since predicting the labels of identifier (reference) tokens is treated as a clustering problem, we can use triplet loss [35]. To compute the loss of a reference node y, we select the true declaration node z and sample a negative declaration node x; the loss is then defined as follows:

$$\mathcal{L}_i(y) = \max(\mathbf{s}^{yx} - \mathbf{s}^{yz}, 0) \tag{25}$$

We can then combine all of the separate components to form a single reconstruction loss function for a node:

$$\mathcal{L}_{rec}(y) = \begin{cases} \mathcal{L}_a(y) + \mathcal{L}_f(y) + \mathcal{L}_r(y), & \text{if } y \text{ is a declaration} \\ \mathcal{L}_a(y) + \mathcal{L}_f(y) + \mathcal{L}_r(y) + \mathcal{L}_i(y), & \text{if } y \text{ is a reference} \\ \mathcal{L}_a(y) + \mathcal{L}_f(y) + \mathcal{L}_r(y) + \mathcal{L}_l(y), & \text{otherwise} \end{cases} \tag{26}$$

Because the loss is decoupled, this allows us to weigh the objectives differently to emphasize, for example, topology or label prediction accuracy. We leave experimenting with different weights for objectives as future work.

The total loss function, combining the KL divergence, KL weight w and reconstruction loss becomes:

$$\mathcal{L}(N) = \mathcal{L}_{tot\_rec}(N) = \sum_{y \in N} \mathcal{L}_{rec}(y) - w \cdot D_{KL}\left(Q(z|N)||P(N)\right) \tag{27}$$

During training, we perform teacher forcing, technique that is commonly used with sequence generation.

3 Evaluation

3.1 Dataset

We train and evaluate our model on a dataset of programs from code competition websites. Programs from these platforms exhibit a few qualities that are suitable for program synthesis. The programs are tested and known to be syntactically correct and compile-able, and they are standalone code fragments and do not depend on any code that is not built into the programming language. The dataset consists of almost two million C++ programs across 148 competitions divided over 904 problems.

The programs in the data set are generally structured to contain a main function, standard input and output stream elements, computation and memory optimizations, and possibly some other elements such as helper functions/classes. Due to this general structure, the programs tend to overlap in their content, in contrast to, for example, natural language sentences.

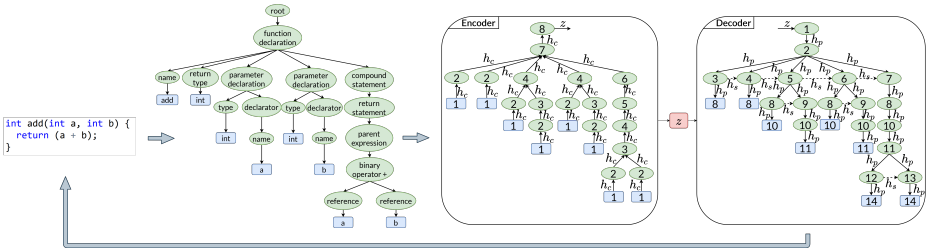

Fig. 3: Tree to tree autoencoder overview. **First Fig.**: The piece of code considered. **Second Fig.**: The piece of code parsed to an AST tree format. **Third Fig.**: The order in which the encoder module encodes the tree structure bottom-up. Here, h_c indicates the hidden state that travels from a child to a parent. **Fourth Fig.**: The order in which the decoder module decodes the tree structure top-down. Here, h_p indicates the hidden state that travels from a parent to a child, and h_s indicates the hidden state that travels from a node to its successor sibling.

3.2 Baseline

Our method is compared to a baseline inspired by autoencoders used for text generation in natural language. We can also evaluate how well these models generalize to source code synthesis by taking inspiration from natural language models. The model architecture is inspired from [36]. In this architecture, both the encoder and decoder networks contain single-layer recurrent neural networks. A Gaussian prior is used for the regularization of the latent space. The model operates on the original sequences of source code and decodes the latent vector back to the source code without an intermediate structured representation. Therefore we refer to the baseline model as the Sequence-to-Sequence (Seq2Seq) model, and the architecture is depicted in fig. 4.

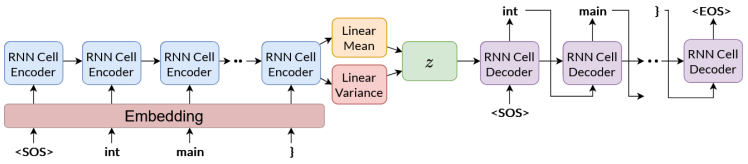

Fig. 4: The architecture of the Seq2Seq model

Similar to our proposed autoencoder model, we employ methods to mitigate KL-vanishing. Again, we use cyclic KL annealing [33], and we combine this with a technique called word dropout [36] to weaken the decoder.

We use three-layered LSTMs in the encoder and decoder with a recurrent dropout rate of 20% to reduce over-fitting. The embedding layer is initialized with Glove wiki gigaword 50 [37] embedding. We train the model using the Adam

optimizer [38] with a learning rate of $1e-3$ and 10 epochs with early stopping and a patience of 3. We train and run the experiments on GPUs with a batch size set to 32.

3.3 Reconstruction results

First of all, we look at how accurately the autoencoders can reconstruct programs. We use a separate test split containing around 60.000 samples of our data set to evaluate this and use these samples as input for the autoencoders.

We compute BLEU scores [39] for both models on the original representation of the source code to obtain comparable results, i.e., we do not use the tree representation. The Tree2Tree model thus has an extra step to use the data parser to transform the tree representation back to source code. This extra step is disadvantageous for the Tree2Tree model as it may introduce some errors due to imperfections in the parsing process. The BLEU scores are then computed on each token in a program: keywords, identifiers, operators, and special symbols such as semicolons or braces. We report on the cumulative BLEU-1 through BLEU-4 scores to indicate the overlap between original and reconstructed programs. Furthermore, we present the percentage of reconstructed programs that compile to indicate how well the models have learned the programming language's syntax. We experiment with different combinations of latent sizes l and hidden RNN dimensions h: (l:10, h:50), (l:50, h:100), (l:100, h:200), (l:150, h:300), (l:300, h:500), (l:500, h:800), (l:800, h:1200).

We use greedy decoding in reconstruction experiments (table table 1) to maximize accuracy. In contrast, sampling is used in generation tasks where diversity of candidates can be helpful.

The results listed in table 1 show the superiority of the Tree2Tree model in terms of reconstruction capability (BLEU scores), especially for smaller latent sizes. The reconstruction scores of the Tree2Tree model of latent size 150 outperform all the Seq2Seq models up to latent size 800. In contrast, the Seq2Seq models show to perform much better at constructing compile-able programs, which improves with the model's size, to nearly 100%. This is a surprising result, which is investigated in more detail in section 3.

An interesting result is that the performance of the models does not necessarily increase with the size of the model. Especially for the Tree2Tree models, we see that after latent size 150, the models' performance decreases. In general, one would expect that the model would perform better with an increase in latent size, allowing more information flow between the encoder and decoder. We hypothesize that, because not only the latent size increases but also the number of hidden units in the auto-regressive models, the models experience KL vanishing. Due to the increasing hidden units, the auto-regressive models become stronger and may depend more on their predictions, ignoring information from the latent vector. In turn, the reconstruction performance vastly decreases. Confirmation of this hypothesis is left as a venue for future work.

Next, we would like to experiment on how different input sizes affect the performance of both models. Due to the tree-structured representation used by

	Latent	BLEU-1	BLEU-2	BLEU-3	BLEU-4	Compiles
Seq	10	0.037	0.024	0.017	0.013	0.000%
	50	0.085	0.061	0.047	0.037	42.467%
	100	0.295	0.225	0.176	0.141	65.808%
	150	0.278	0.211	0.165	0.131	66.971%
	300	0.346	0.262	0.203	0.161	60.651%
	500	0.421	0.332	0.263	0.211	90.329%
	800	0.429	0.329	0.253	0.195	**91.784%**
Tree	10	0.445	0.339	0.260	0.202	28.375%
	50	0.417	0.317	0.242	0.189	23.256%
	100	0.423	0.323	0.251	0.200	30.429%
	150	**0.486**	**0.382**	**0.302**	**0.243**	35.419%
	300	0.457	0.342	0.260	0.202	35.054%
	500	0.398	0.301	0.230	0.178	36.022%
	800	0.258	0.182	0.131	0.096	2.358%

Table 1: Reconstruction results.

the Tree2Tree model, the size of the sequences that the RNNs process scale proportionally to the width and depth of the tree. The Seq2Seq model, on the other hand, processes sequences left to right, hence the number of computations of the RNNs scale directly with the sequence length.

To evaluate the performance on different sized inputs, we split the test data set into three subsets. A small, medium and large subset with the following properties:

- **small subset**: maximum of 250 tokens
- **medium subset**: between 251 and 500 tokens
- **large subset**: between 501 and 750 tokens

We compute the BLEU scores and compilation percentage again using greedy decoding on the smaller subsets for the best performing Seq2Seq and Tree2Tree models, based on the results of table 1. Here, performance is based on the combination of BLEU-4 and compilation percentage. For Seq2Seq, this is the model with latent size 500. Similarly, for Tree2Tree, this is the model with latent size 150. The results are depicted in table 2.

From table 2 we can observe that both models follow the same logical trend: the larger the input size, the lower BLEU-scores and compilation percentages. For the Tree2Tree model, the BLEU scores for the medium subset seem to be similar to the BLEU scores on the entire test set, whereas, for the Seq2Seq model, the BLEU scores are much lower on the medium subset. The models

	Input	BLEU-1	BLEU-2	BLEU-3	BLEU-4	Compiles
Seq	small	0.513	0.403	0.321	0.258	95.334%
	medium	0.306	0.244	0.192	0.153	86.812%
	large	0.196	0.157	0.123	0.096	87.971%
Tree	small	0.633	0.516	0.424	0.355	59.022%
	medium	0.478	0.371	0.289	0.229	21.241%
	large	0.324	0.242	0.181	0.138	5.001%

Table 2: Reconstruction results of the best models on different input sizes.

seem to be fairly close in terms of performance degradation from small to large program sizes. For example, we can measure performance degradation for the large versus small subset by dividing the BLEU-4 scores on the large set by the BLEU-4 score on the small set. For the Seq2Seq model, we get a score of 0.372, and for the Tree2Tree model, we get 0.389. Similarly, we get 0.593 and 0.645 for the Seq2Seq and Tree2Tree model for the medium versus small subset. While the performance degrades less with increasing input sizes for the Tree2Tree model, this difference is insignificant.

An issue with the aforementioned computation of performance degradation is that it does not correct for elements in programs that are almost always present. For example, each program contains a main function, with standard input and output streams. The models may simply always predict these standard elements of a program and then use the information of the encoder to complete the details of the program. However, this causes the BLEU score to consist of two parts: the score for the prediction of the elements that are always present and the score of what it has learned to predict together with the encoder. The latter is more interesting and shows how much information can be saved in the latent vector.

Therefore, we apply a correction on the BLEU scores to focus on the prediction based on the information in the latent vector. We compute corrected scores by feeding the decoder with random latent vectors and computing BLEU scores on the subsets of the test data set. Then, we subtract these correction scores from the computed BLEU scores in table 2, and take 0 if the result of the subtraction is negative. The corrected BLEU scores including the correction scores are presented in table 3.

Table 3 indicates a large difference in performance degradation between the Seq2Seq model and the Tree2Tree model. A noticeable result is that the corrected BLEU scores for large programs predicted by the Seq2Seq model are 0. Hence, the Seq2Seq model extracts no information from the latent vector at all for large programs. Similarly, for medium-sized programs, little information is transferred between the encoder and decoder. We can again compute the performance degradation scores for the Seq2Seq model, which are 0.280 and 0.00 for the medium versus small and large versus small subsets, respectively, on the corrected BLEU-4.

Model	Input size	BLEU-1	BLEU-2	BLEU-3	BLEU-4
Seq2Seq	small	0.072 (0.441)	0.077 (0.326)	0.075 (0.246)	0.070 (0.188)
	medium	0.006 (0.300)	0.018 (0.226)	0.021 (0.171)	0.023 (0.130)
	large	0.000 (0.213)	0.000 (0.166)	0.000 (0.128)	0.000 (0.099)
Tree2Tree	small	0.200 (0.433)	0.220 (0.296)	0.223 (0.201)	0.218 (0.137)
	medium	0.148 (0.330)	0.147 (0.224)	0.146 (0.150)	0.128 (0.101)
	large	0.102 (0.222)	0.090 (0.152)	0.079 (0.102)	0.070 (0.068)

Table 3: Corrected BLEU scores of reconstructed results of the best models on different input sizes. (correction scores in parenthesis)

In contrast, the performance degradation is much smaller for the Tree2Tree model: 0.587 and 0.321 for the medium versus small and large versus small subsets, respectively, on the corrected BLEU-4. Hence, the structural nature of the Tree2Tree model scales better to large input sequences than the Seq2Seq model in terms of reconstruction scores, even with a much smaller latent size.

An interesting observation is that the latent vector conveys relatively little information in terms of BLEU scores. The correction scores make up a large part of the total BLEU scores as presented in table 2. Hence, the BLEU scores are largely determined by the models' general knowledge of how C++ programs are built up and not the specific content.

3.4 Generative results

To see how well autoencoders can generate reasonable samples from any point in latent space that conform to the C++ syntax, we sample 1000 random latent vectors from the prior distribution $\mathcal{N}(0, I)$ and input these vectors to the decoder networks. Then, we compute the percentage of generated programs that compiles and is thus also syntactically correct.

We employ two decoding strategies to test the generative capabilities of the models: greedy decoding and sampling. The sampling strategy we apply is a combination of top-k, nucleus, and temperature sampling [40]. We first use temperature sampling to scale the logits to control the shape of the probability distribution. Then, we filter the on the top-k samples, after which we filter tokens on their cumulative probability using nucleus sampling (top-p). Lastly, we sample a token from the resulting distribution. The selected sampling hyperparameters for this experiment are: $k = 40$, $p = 0.9$, $temperature = 0.7$. The results of the experiment are displayed in table 4.

The results from table 4 show similar trends as section 3.3. The general trend is: the larger the model (in terms of latent size and hidden units), the higher the compilation percentage. Moreover, greedy search during inference gives a higher compilation percentage than sampling. This outcome is not surprising, as, with greedy search, we always pick the label for which the model is most confident.

Model	Latent size	Greedy search	Sampling
Seq2Seq	10	0.0%	0.9%
	50	38.5%	2.9%
	100	62.1%	21.3%
	150	58.0%	23.5%
	300	60.6%	36.8%
	500	67.5%	37.8%
	800	78.2%	39.6%
Tree2Tree	10	29.6%	20.2%
	50	22.6%	17.7%
	100	30.3%	22.1%
	150	26.9%	18.8%
	300	23.4%	12.8%
	500	25.6%	14.4%
	800	4.1%	6.7%

Table 4: Generative results compilation percentage.

On the other hand, sampling gives a more varied output and may be useful for searching similar programs in a vicinity of the latent space. The trade-off for a more diverse output is thus a lower compilation ratio.

4 Conclusion

Our results indicate that Tree Variational Autoencoders have a significant advantage over sequence-to-sequence models in low-dimensional latent spaces, achieving both a higher compilation rate and a higher reconstruction quality. In higher-dimensional latent spaces seq2seq programs offer a higher compilation rate, but based corrected BLEU scores indicate that this benefit is often achieved by sacrificing reconstruction quality, even to the point of ignoring the input completely.

Overall, our findings support the initial hypothesis that structured autoencoder models are better suited for program synthesis than sequence-to-sequence alternatives. We believe this result to be a significant step towards an autoencoder-based foundation model for genetic programming and genetic improvement of software.

References

1. R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill *et al.*, "On the opportunities and risks of foundation models," *arXiv preprint arXiv:2108.07258*, 2021.

2. A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, "Improving language understanding by generative pre-training," 2018.
3. M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.
4. E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *ArXiv preprint, abs/2203.13474*, 2022.
5. S. Black, S. Biderman, E. Hallahan, Q. Anthony, L. Gao, L. Golding, H. He, C. Leahy, K. McDonell, J. Phang *et al.*, "Gpt-neox-20b: An open-source autoregressive language model," *arXiv preprint arXiv:2204.06745*, 2022.
6. J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.
7. Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.
8. D. C. Halbert, "Programming by example," Ph.D. dissertation, University of California, Berkeley, 1984.
9. V. Liventsev, "The unreasonable effectiveness of language models for source code," Aug 2022. [Online]. Available: https://vadim.me/publications/unreasonable/
10. Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.
11. J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*. MIT press, 1992, vol. 1.
12. J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward, "Genetic improvement of software: a comprehensive survey," *IEEE Transactions on Evolutionary Computation*, vol. 22, no. 3, pp. 415–432, 2017.
13. J. McDermott, "Why is auto-encoding difficult for genetic programming?" in *European Conference on Genetic Programming*. Springer, 2019, pp. 131–145.
14. D. Wittenberg, "Using denoising autoencoder genetic programming to control exploration and exploitation in search," in *European Conference on Genetic Programming (Part of EvoStar)*. Springer, 2022, pp. 102–117.
15. P. Liskowski, K. Krawiec, N. E. Toklu, and J. Swan, "Program synthesis as latent continuous optimization: evolutionary search in neural embeddings," in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, 2020, pp. 359–367.
16. D. Bank, N. Koenigstein, and R. Giryes, "Autoencoders," *arXiv preprint arXiv:2003.05991*, 2020.
17. U. Alon, R. Sadaka, O. Levy, and E. Yahav, "Structural language models for any-code generation," *arXiv preprint arXiv:1910.00577*, 2019.
18. X. Zhang, L. Lu, and M. Lapata, "Tree recurrent neural networks with application to language modeling," *CoRR, abs/1511.00060*, 2015.
19. H. Jiang, L. Song, Y. Ge, F. Meng, J. Yao, and J. Su, "An ast structure enhanced decoder for code generation," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 30, pp. 468–476, 2021.
20. Z. Sun, Q. Zhu, L. Mou, Y. Xiong, G. Li, and L. Zhang, "A grammar-based structural cnn decoder for code generation," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, pp. 7055–7062, Jul. 2019. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/4686

21. X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," *arXiv preprint arXiv:1802.03691*, 2018.
22. M. J. Kusner, B. Paige, and J. M. Hernández-Lobato, "Grammar variational autoencoder," in *International Conference on Machine Learning*. PMLR, 2017, pp. 1945–1954.
23. L. R. Kosta Jr, "Program synthesis and vulnerability injection using a grammar vae," Ph.D. dissertation, Boston University, 2019.
24. D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013.
25. K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," *arXiv preprint arXiv:1503.00075*, 2015.
26. A. Graves, N. Jaitly, and A.-r. Mohamed, "Hybrid speech recognition with deep bidirectional lstm," in *2013 IEEE workshop on automatic speech recognition and understanding*. IEEE, 2013, pp. 273–278.
27. G. I. Winata, O. P. Kampman, and P. Fung, "Attention-based lstm for psychological stress detection from spoken language using distant supervision," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2018, pp. 6204–6208.
28. O. Fabius and J. R. van Amersfoort, "Variational recurrent auto-encoders," 2015.
29. C.-C. Kao, M. Sun, W. Wang, and C. Wang, "A comparison of pooling methods on lstm models for rare acoustic event classification," 2020.
30. D. Alvarez-Melis and T. S. Jaakkola, "Tree-structured decoding with doubly-recurrent neural networks," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2017.
31. M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 25–36.
32. E. Grave, A. Joulin, M. Cissé, D. Grangier, and H. Jégou, "Efficient softmax approximation for gpus," 2017.
33. H. Fu, C. Li, X. Liu, J. Gao, A. Celikyilmaz, and L. Carin, "Cyclical annealing schedule: A simple approach to mitigating kl vanishing," *arXiv preprint arXiv:1903.10145*, 2019.
34. T. Long, Y. Cao, and J. C. K. Cheung, "Preventing posterior collapse in sequence vaes with pooling," *arXiv preprint arXiv:1911.03976*, 2019.
35. G. Chechik, V. Sharma, U. Shalit, and S. Bengio, "Large scale online learning of image similarity through ranking." *Journal of Machine Learning Research*, vol. 11, no. 3, 2010.
36. S. R. Bowman, L. Vilnis, O. Vinyals, A. M. Dai, R. Jozefowicz, and S. Bengio, "Generating sentences from a continuous space," *arXiv preprint arXiv:1511.06349*, 2015.
37. J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.
38. D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.
39. K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

40. A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The curious case of neural text degeneration," *arXiv preprint arXiv:1904.09751*, 2019.