Machine Learning (2022) 111:3393–3423
https://doi.org/10.1007/s10994-022-06223-7

Recursive tree grammar autoencoders

Benjamin Paaßen[1,2] · Irena Koprinska[1] · Kalina Yacef[1]

Received: 16 February 2022 / Revised: 24 May 2022 / Accepted: 2 July 2022 /
Published online: 2 August 2022

Abstract
Machine learning on trees has been mostly focused on trees as input. Much less research has investigated trees as output, which has many applications, such as molecule optimization for drug discovery, or hint generation for intelligent tutoring systems. In this work, we propose a novel autoencoder approach, called recursive tree grammar autoencoder (RTG-AE), which encodes trees via a bottom-up parser and decodes trees via a tree grammar, both learned via recursive neural networks that minimize the variational autoencoder loss. The resulting encoder and decoder can then be utilized in subsequent tasks, such as optimization and time series prediction. RTG-AEs are the first model to combine three features: recursive processing, grammatical knowledge, and deep learning. Our key message is that this unique combination of all three features outperforms models which combine any two of the three. Experimentally, we show that RTG-AE improves the autoencoding error, training time, and optimization score on synthetic as well as real datasets compared to four baselines. We further prove that RTG-AEs parse and generate trees in linear time and are expressive enough to handle all regular tree grammars.

Keywords Recursive neural networks · Tree grammars · Representation learning · Variational autoencoders

Editors: Krzysztof Dembczynski, Emilie Devijver.

✉ Benjamin Paaßen
benjamin.paassen@dfki.de

Irena Koprinska
irena.koprinska@sydney.edu.au

Kalina Yacef
kalina.yacef@sydney.edu.au

1 School of Computer Science, The University of Sydney, 1 Cleveland Street, Darlington, NSW 2008, Australia

2 Educational Technology Lab, German Research Center for Artificial Intelligence (DFKI), Alt-Moabit 91c, 10559 Berlin, Berlin, Germany

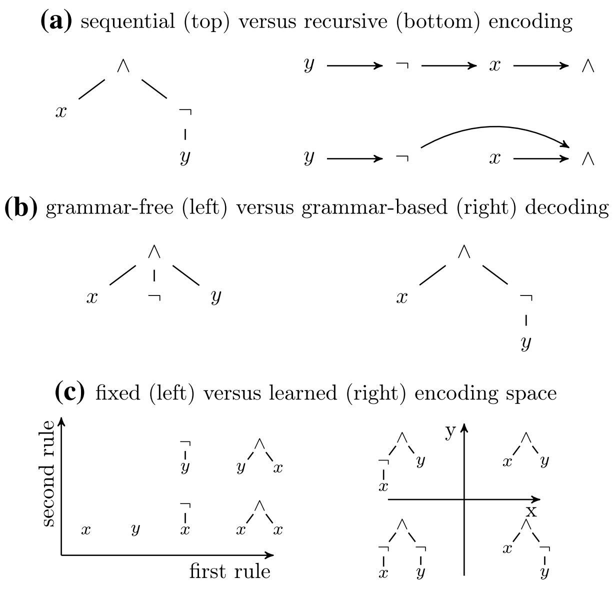
(a) sequential (top) versus recursive (bottom) encoding
(b) grammar-free (left) versus grammar-based (right) decoding
(c) fixed (left) versus learned (right) encoding space
second rule
first rule
y
x

Fig. 1 An illustration of the advantages (**a**) of recursive over sequential processing, **b** of utilizing grammatical knowledge, and **c** of learning the encoding end-to-end. In (**c**), each point represents the encoding of a tree and color indicates some semantic attribute with respect to which the encoding space should be smooth (right)

1 Introduction

Neural networks on trees have made significant progress in recent years, achieving unprecedented performance with new models such as tree echo state networks (Gallicchio & Micheli, 2013), tree LSTMs (Tai et al., 2015), code2vec (Alon et al., 2019), or models from the graph neural network family (Kipf & Welling, 2017; Micheli, 2009; Scarselli et al., 2009). However, these achievements are mostly limited to tasks with *numeric output*, such as classification and regression. By contrast, much less research has focused on tasks that require *trees as output*, such as molecular design (Kusner et al., 2017; Jin et al., 2018) or hint generation in intelligent tutoring systems (Paaßen et al., 2018). For such tasks, autoencoder models are particularly attractive because they support both encoding and decoding, thus enabling tree-to-tree tasks (Kusner et al., 2017). In this paper, we propose a novel autoencoder for trees, which is the first model to combine grammar information, recursive processing, and deep learning. Hence, we name it recursive tree grammar autoencoder (RTG-AE).

Our core claim in this paper is that combining these three features—recursive processing, grammar knowledge, and deep learning—performs better than combining only two of the three (refer to Fig. 1). Incidentally, there exist baseline models in the literature which combine two features but not the last one. In particular, the grammar variational autoencoder (Kusner et al., 2017, GVAE) represents strings as a sequence of context-free grammar rules and is trained via deep learning, but does not use recursive processing. Instead, it encodes the rule sequence via a series of 1D convolutions and a fully connected layer; and it decodes a vector back to a rule sequence via a multi-layer gated recurrent unit (Cho et al., 2014, GRU). We believe that this sequential scheme makes sense for string data but becomes a limitation for trees because the sequential processing introduces long-term dependencies that do not exist in the tree. For example, when processing the tree $\wedge(x, \neg(y))$, a sequential representation would be $y, \neg, x, \wedge$. When processing $\wedge$, we want to take the information from its children x and

$\neg$ into account, but $\neg$ is already two steps away (refer to Fig. 1a). If we replace x with a large subtree, this distance can become arbitrarily large.

Recursive neural networks (Tai et al., 2015; Pollack, 1990; Sperduti & Starita, 1997) avoid this problem. They compute the representation of a parent node based on the representation of all children, meaning the information flow follows the tree structure and the length of dependencies is bounded by the depth of the tree. There also exist autoencoding models in the recursive neural network tradition, such as the model of Pollack (1990) or the directed acyclic graph variational autoencoder (M. Zhang, Jiang, Cui, Garnett, & Chen, 2019, D-VAE). Unfortunately, the autoencoding capability of recursive networks is limited due to the enormous number of trees one *could* potentially decode from a vector. More precisely, Hammer (2002) showed that one needs exponentially many neurons to represent all trees of a certain depth with a recursive network. We believe that grammars help to avoid this limitation because the set of grammatical trees is usually much smaller than the set of possible trees over an alphabet. For example, the tree $\wedge(x, \neg(y))$ represents a valid Boolean expression but not the tree $\wedge(x, \neg, y)$ (refer to Fig. 1b). Without a grammar as inductive bias, models like D-VAE need to learn to avoid trees such as $\wedge(x, \neg, y)$. D-VAE serves as our baseline model which uses recursive processing and deep learning but no grammar knowledge.

Finally, in prior research we proposed tree echo state auto encoders (Paaßen, Koprinska, & Yacef, 2020, TES-AE), a recursive, grammar-based autoencoder for trees which does not use deep learning. Instead, this model randomly initializes its network parameters and only trains the final layer which chooses the next grammar rule for decoding. This shallow learning scheme follows the (tree) echo state network paradigm, which claims that a sufficiently large, randomly wired network is expressive enough to represent any input (Gallicchio & Micheli, 2013). However, a fixed representation may need more dimensions compared to one that adjusts to the data. Consider our example of Boolean formulae, again. Let's code x as 0, y as 1, $\neg$ as 2, and $\wedge$ as 3. We can then encode trees as sequences of these numbers, padding with zeros wherever needed. In 2D, we can then represent all trees with up to three nodes (filling up with x, where needed). In particular, (0, 0) corresponds to x, (1, 0) to y, (2, 0) to $\neg(x)$, (3, 0) to $\wedge(x, x)$, (2, 1) to $\neg(y)$, and (3, 1) to $\wedge(y, x)$. However, we can also adapt our encoding by using the first dimension to encode the x variable, and the second dimension to encode the y variable, such that (1, 0) decodes to x, (0, 1) to y, (1, 1) to $\wedge(x, y)$, and $(1, -1)$ to $\wedge(x, \neg(y))$ (refer to Fig. 1c). Adjusting to the data enabled us to represent larger trees with the same number of dimensions and to better take the semantics of the domain into account. Further, learning enables us to enforce smoothness in the coding space, which may be helpful for downstream tasks.

The key contributions of our work are:

- We develop a novel autoencoder for trees which is the first to combine recursive processing, grammar knowledge, and deep learning, whereas prior models combined only two of the three. We call our model RTG-AE.
- We provide a correctness proof for the encoding scheme of RTG-AE.
- Experimentally, we compare RTG-AE to models which combine two of the three features, namely GVAE, which combines grammar knowledge and deep learning, but not recursive processing; D-VAE, which combines recursive processing and deep learning, but not grammar knowledge; and TES-AE, which combines recursive processing and grammar knowledge, but not deep learning. We observe that RTG-AE has the lowest autoencoding error and runtime—except for TES-AE, which has

lower autoencoding error on the smallest dataset and is always faster because it does not use deep learning.

- In a further experiment, we evaluate the capability of a CMA-ES optimizer to find an optimal tree in the encoding space of each model. We find that RTG-AE yields the best median scores.

We begin by discussing background and related work before we introduce the RTG-AE architecture and evaluate it on four datasets, including two synthetic and two real-world ones.

2 Background and related work

Our contribution relies on substantial prior work, both from theoretical computer science and machine learning. We begin by introducing our formal notion of trees and tree grammars, after which we continue with neural networks for tree representations.

2.1 Regular tree grammars

Let Σ be some finite alphabet of symbols. We recursively define a *tree* $\hat{x}$ over Σ as an expression of the form $\hat{x} = x(\hat{y}_1, \dots, \hat{y}_k)$, where $x \in \Sigma$ and where $\hat{y}_1, \dots, \hat{y}_k$ is a list of trees over Σ. We call $\hat{x}$ a *leaf* if $k = 0$, otherwise we call $\hat{y}_1, \dots, \hat{y}_k$ the *children* of $\hat{x}$. We define the *size* $|\hat{x}|$ of a tree $\hat{x} = x(\hat{y}_1, \dots, \hat{y}_k)$ as $1 + |\hat{y}_1| + \dots + |\hat{y}_k|$.

Next, we define a *regular tree grammar* (RTG) (Brainerd, 1969; Comon et al., 2008) $\mathcal{G}$ as a 4-tuple $\mathcal{G} = (\Phi, \Sigma, R, S)$, where Φ is a finite set of nonterminal symbols, Σ is a finite alphabet as before, $S \in \Phi$ is a special nonterminal symbol which we call the starting symbol, and R is a finite set of production rules of the form $A \to x(B_1, \dots, B_k)$ where $A, B_1, \dots, B_k \in \Phi$, $k \in \mathbb{N}_0$, and $x \in \Sigma$. We say a sequence of rules $r_1, \dots, r_T \in R$ *generates* a tree $\hat{x}$ from some nonterminal $A \in \Phi$ if applying all rules to A yields $\hat{x}$, as specified in Algorithm 1. We define the regular tree language $\mathcal{L}(\mathcal{G})$ of grammar $\mathcal{G}$ as the set of all trees that can be generated from S via some (finite) rule sequence over R.

Algorithm 1 An algorithm for generating a tree $\hat{x}$ from a nonterminal S and a sequence of production rules $r_1, \ldots, r_T$.

```
function GENERATE_TREE(nonterminal $S$, rules $r_1, \ldots, r_T$)
    Initialize a stack $\mathcal{S}$ with $S$ on top.
    Initialize a tree $\hat{x} \leftarrow S()$.
    for $t \leftarrow 1, \ldots, T$ do
        Pop nonterminal $A'$ from $\mathcal{S}$.
        Let $r_t = A \rightarrow x(B_1, \ldots, B_k)$.
        if $A \neq A'$ then
            Process fails.
        end if
        Replace $A'$ in the tree with $x(B_1, \ldots, B_k)$.
        Push $B_k, \ldots, B_1$ onto $\mathcal{S}$.
    end for
    if $\mathcal{S}$ is not empty then
        Process fails.
    end if
    return $\hat{x}$.
end function
```

The inverse of generation is called *parsing*. In our case, we rely on the bottom-up parsing approach of Comon et al. (2008), as shown in Algorithm 2. For the input tree $\hat{x} = x(\hat{y}_1, \ldots, \hat{y}_k)$, we first parse all children, yielding a nonterminal B_j and a rule sequence $\bar{r}_j$ that generates child $\hat{y}_j$ from B_j. Then, we search the rule set R for a rule of the form $r = A \rightarrow x(B_1, \ldots, B_k)$ for some nonterminal A, and finally return the nonterminal A as well as the rule sequence $r, \bar{r}_1, \ldots, \bar{r}_k$, where the commas denote concatenation. If we don't find a matching rule, the process fails. Conversely, if the algorithm returns successfully, this implies that the rule sequence $r, \bar{r}_1, \ldots, \bar{r}_k$ generates $\hat{x}$ from A. Accordingly, if $A = S$, then $\hat{x} \in \mathcal{L}(\mathcal{G})$.

Algorithm 2 An algorithm for parsing a tree $\hat{x}$ according to a regular tree grammar $\mathcal{G} = (\Phi, \Sigma, R, S)$.

```
function PARSE_TREE(tree $\hat{x} = x(\hat{y}_1, \ldots, \hat{y}_k)$)
    for $j \leftarrow 1, \ldots, k$ do
        $B_j, \bar{r}_j \leftarrow$ PARSE_TREE($\hat{y}_k$).
    end for
    if $\exists A \in \Phi : A \rightarrow x(B_1, \ldots, B_k) \in R$ then
        $r \leftarrow A \rightarrow x(B_1, \ldots, B_k)$.
        return $A$, $(r, \bar{r}_1, \ldots, \bar{r}_k)$.
    else
        Process fails.
    end if
end function
```

Algorithm 2 can be ambiguous if multiple nonterminals A exist such that $A \to x(B_1, \dots, B_k) \in R$ in line 5. To avoid such ambiguities, we impose that our regular tree grammars are *deterministic*, i.e. no two grammar rules have the same right-hand-side. This is sufficient to ensure that any tree corresponds to a unique rule sequence.

Theorem 1 (mentioned in page 24 of Comon et al. (2008)) *Let $\mathcal{G} = (\Phi, \Sigma, R, S)$ be a regular tree grammar. Then, for any $\hat{x} \in \mathcal{L}(\mathcal{G})$ there exists exactly one sequence of rules $r_1, \dots, r_T \in R$ which generates $\hat{x}$.*

Proof Refer to Appendix A.1. □

This is no restriction to expressiveness, as any regular tree grammar can be transformed into an equivalent, deterministic one.

Theorem 2 (Therorem 1.1.9 by Comon et al. (2008)) *Let $\mathcal{G} = (\Phi, \Sigma, R, S)$ be a regular tree grammar. Then, there exists a regular tree grammar $\mathcal{G}' = (\Phi', \Sigma, R', \mathcal{S})$ with a set of starting symbols $\mathcal{S}$ such that $\mathcal{G}'$ is deterministic and $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{G}')$.*

Proof Refer to Appendix A.2. □

It is often convenient to permit two further concepts in a regular tree grammar, namely optional and starred nonterminals. In particular, the notation $B?$ denotes a nonterminal with the production rules $B? \to B$ and $B \to \varepsilon$, where ε is the empty word. Similarly, B^* denotes a nonterminal with the production rules $B^* \to B, B^*$ and $B^* \to \varepsilon$. To maintain determinism, one must ensure two conditions: First, if a rule generates two adjacent nonterminals that are starred or optional, then these nonterminals must be different, so $A \to x(B\,*, C\,*)$ is permitted but $A \to x(B\,*, B?)$ is not, because we would not know whether to assign an element to $B\,*$ or $B?$. Second, the languages generated by any two right-hand-sides for the same nonterminal must be non-intersecting. For example, if the rule $A \to x(B\,*, C)$ exists, then the rule $A \to x(C?, D\,*)$ is not allowed because the right-hand-side $x(C)$ could be generated by either of them (refer to Appendix A.2 for more details). In the remainder of this paper, we generally assume that we deal with deterministic regular tree grammars that may contain starred and optional nonterminals.

2.2 Tree encoding

We define a *tree encoder* for a regular tree grammar $\mathcal{G}$ as a mapping $\phi : \mathcal{L}(\mathcal{G}) \to \mathbb{R}^n$ for some encoding dimensionality $n \in \mathbb{N}$. While fixed tree encodings do exist, e.g. in the form of tree kernels (Aiolli et al.., 2015; Collins & Duffy, 2002), we focus here on *learned* encodings via deep neural networks. A simple tree encoding scheme is to list all nodes of a tree in depth-first-search order and encode this list via a recurrent or convolutional neural network (Paaßen et al., 2020). However, one can also encode the tree structure more directly via recursive neural networks (Gallicchio & Micheli, 2013; Tai et al., 2015; Pollack, 1990; Sperduti & Starita, 1997; Sperduti, 1994). Generally speaking, a recursive neural network consists of a set of mappings $f^x : \mathcal{P}(\mathbb{R}^n) \to \mathbb{R}^n$, one for each symbol $x \in \Sigma$, which receive a (perhaps ordered) set of child encodings as input and map it to a parent encoding. Based on such mappings, we define the overall tree encoder recursively as

$$\phi\big(x(\hat{y}_1, \ldots, \hat{y}_k)\big) := f^x\big(\{\phi(y_1), \ldots, \phi(y_k)\}\big). \tag{1}$$

Traditional recursive neural networks implement f^x with single- or multi-layer perceptrons. More recently, recurrent neural networks have been applied, such as echo state nets (Gallicchio & Micheli, 2013) or LSTMs (Tai et al., 2015). In this work, we extend the encoding scheme by defining the mappings f^x not over terminal symbols x but over grammar rules r, thereby tying encoding closely to parsing. This circumvents a typical problem in recursive neural nets, namely to handle the order and number of children (Sperduti & Starita, 1997).

Recursive neural networks can also be related to more general graph neural networks (Kipf & Welling, 2017; Micheli, 2009; Scarselli et al., 2009). In particular, we can interpret a recursive neural network as a graph neural network which transmits messages from child nodes to parent nodes until the root is reached. Thanks to the acyclic nature of trees, a single pass from leaves to root is sufficient, whereas most graph neural net architectures would require as many passes as the tree is deep (Kipf & Welling, 2017; Micheli, 2009; Scarselli et al., 2009). In other words, graph neural nets only consider neighboring nodes in a pass, whereas recursive nets incorporate information from all descendant nodes. Another reason why we choose to consider trees instead of general graphs is that graph grammar parsing is NP-hard (Turán, 1983), whereas regular tree grammar parsing is linear (Comon et al., 2008).

For the specific application of encoding syntax trees of computer programs, three further strategies have been proposed recently, namely: Code2vec considers paths from the root to single nodes and aggregates information across these paths using attention (Alon et al., 2019); AST-NN treats a syntax tree as a sequence of subtrees and encodes these subtrees first, followed by a GRU which encodes the sequence of subtree encodings (Zhang et al., 2019); and CuBERT treats source code as a sequence of tokens which are then plugged into a big transformer model from natural language processing (Kanade et al., 2020). Note that these models focus on encoding trees, whereas we wish to support encoding as well as decoding.

2.3 Tree decoding

We define a *tree decoder* for a regular tree grammar $\mathcal{G}$ as a mapping $\psi : \mathbb{R}^n \rightarrow \mathcal{L}(\mathcal{G})$ for some encoding dimensionality $n \in \mathbb{N}$. In early work, Pollack (1990) and Sperduti (1994) already proposed decoding mechanisms using 'inverted' recursive neural networks, i.e. mapping from a parent representation to a fixed number of children, including a special 'none' token for missing children. Theoretical limits of this approach have been investigated by Hammer (2002), who showed that one requires exponentially many neurons to decode all possible trees of a certain depth. More recently, multiple works have considered the more general problem of decoding graphs from vectors, where a graph is generated by a sequence of node and edge insertions, which in turn is generated via a deep recurrent neural net (Zhang et al., 2019; Bacciu et al., 2019; Liu et al., 2018; Paaßen et al., 2021; You et al., 2018). From this family, the variational autoencoder for directed acyclic graphs (D-VAE) (Zhang et al., 2019) is most suited to trees because it explicitly prevents cycles. In particular, the network generates nodes one by one and then decides which of the earlier nodes to connect to the new node, thereby preventing cycles. We note that there is an entire branch of graph generation devoted specifically to molecule design which is beyond our capability to cover here (Sanchez-Lengeling & Aspuru-Guzik, 2018). However, tree decoding may serve as a subroutine, e.g. to construct a junction tree in (Jin et al., 2018).

Another thread of research concerns the generation of strings from a context-free grammar, guided by a recurrent neural network (Kusner et al., 2017; Dai et al., 2018). Roughly speaking, these approaches first parse the input string, yielding a generating rule sequence, then convert this rule sequence into a vector via a convolutional neural net, and finally decode the vector back into a rule sequence via a recurrent neural net. This rule sequence, then, yields the output string. Further, one can incorporate additional syntactic or semantic constraints via attribute grammars in the rule selection step (Dai et al., 2018). We follow this line of research but use tree instead of string grammars and employ recursive instead of sequential processing. This latter change is key because it ensures that the distance between the encoding and decoding of a node is bounded by the tree depth instead of the tree size, thus decreasing the required memory capacity from linear to logarithmic in the tree size.

A third thread of research attempts to go beyond known grammars and instead tries to infer a grammar from data, typically using stochastic parsers and grammars that are controlled by neural networks (Allamanis et al., 2017; Dyer et al., 2016; Li et al., 2019; Kim et al., 2019; Yogatama et al., 2017; Zaremba et al., 2014). Our work is similar in that we also control a parser and a grammar with a neural network. However, our task is conceptually different: We assume a grammar is given and are solely concerned with autoencoding trees within the grammar's language, whereas these works attempt to find tree-like structure in strings. While this decision constrains us to known grammars, it also enables us to consider non-binary trees and variable-length rules which are currently beyond grammar induction methods. Further, pre-specified grammars are typically designed to support interpretation and semantic evaluation (e.g. via an objective function for optimization). Such an interpretation is much more difficult for learned grammars.

Finally, we note that our own prior work (Paaßen et al., 2020) already combines tree grammars with recursive neural nets (in particular tree echo state networks (Gallicchio & Micheli, 2013)). However, in this paper we combine such an architecture with an end-to-end-learned variational autoencoder, thus guaranteeing a smooth latent space, a standard normal distribution in the latent space, and smaller latent spaces. It also yields empirically superior results, as we see later in the experiments.

2.4 Variational autoencoders

An autoencoder is a combination of an encoder ϕ and a decoder ψ that is trained to minimize some form of autoencoding error, i.e. some notion of dissimilarity between an input $\hat{x}$ and its autoencoded version $\psi(\phi(\hat{x}))$. In this paper, we consider the variational autoencoder (VAE) approach of Kingma and Welling (2019), which augments the deterministic encoder and decoder to probability distributions from which we can sample. More precisely, we introduce a probability density $q_\phi(\mathbf{z}|\hat{x})$ for encoding $\hat{x}$ into a vector $\mathbf{z}$, and a probability distribution $p_\psi(\hat{x}|\mathbf{z})$ for decoding $\mathbf{z}$ into $\hat{x}$.

Now, let $\hat{x}_1, \dots, \hat{x}_m$ be a training dataset. We train the autoencoder to minimize the loss:

$$\ell(\phi, \psi) = \sum_{i=1}^{m} \mathbb{E}_{q_\phi(z_i|\hat{x}_i)} \Big[-\log \big[p_\psi \big(\hat{x}_i | \mathbf{z}_i \big) \big] \Big] + \beta \cdot \mathcal{D}_{\mathrm{KL}}(q_\phi \| \mathcal{N}), \tag{2}$$

where $\mathcal{D}_{\mathrm{KL}}$ denotes the Kullback-Leibler divergence between two probability densities and where $\mathcal{N}$ denotes the density of the standard normal distribution. β is a hyper-parameter to weigh the influence of the second term, as suggested by Burda et al. (2016).

Typically, the loss in (2) is minimized over tens of thousands of stochastic gradient descent iterations, such that the expected value over $q_\phi(z_i|\hat{x}_i)$ can be replaced with a single sample (Kingma & Welling, 2019). Further, q_ϕ is typically modeled as a Gaussian with diagonal covariance matrix, such that the sample can be re-written as $\mathbf{z}_i = \rho(\boldsymbol{\mu}_i + \boldsymbol{\epsilon}_i \odot \boldsymbol{\sigma}_i)$, where $\boldsymbol{\mu}_i$ and $\boldsymbol{\sigma}_i$ are deterministically generated by the encoder ϕ, where $\odot$ denotes element-wise multiplication, and where $\boldsymbol{\epsilon}_i$ is Gaussian noise, sampled with mean zero and standard deviation s. s is a hyper-parameter which regulates the noise strength we impose during training.

We note that many extensions to variational autoencoders have been proposed over the years (Kingma & Welling, 2019), such as Ladder-VAE (Sønderby et al., 2016) or InfoVAE (Zhao et al., 2019). Our approach is generally compatible with such extensions, but our focus here lies on the combination of autoencoding, grammatical knowledge, and recursive processing, such that we leave extensions of the autoencoding scheme for future work.

3 Method

Our proposed architecture is a variational autoencoder for trees, where we construct the encoder as a bottom-up parser, the decoder as a regular tree grammar, and the reconstruction loss as the crossentropy between the true rules generating the input tree and the rules chosen by the decoder. An example autoencoding computation is shown in Fig. 2. Because our encoding and decoding schemes are closely related to recursive neural networks (Pollack, 1990; Sperduti & Starita, 1997; Sperduti, 1994), we call our approach *recursive tree grammar autoencoders* (RTG-AEs). We now introduce each of the components in turn.

3.1 Encoder

Our encoder is a bottom-up parser for a given regular tree grammar $\mathcal{G} = (\Phi, \Sigma, R, S)$, computing a vectorial representation in parallel to parsing. In more detail, we introduce an encoding function $f^r : \mathbb{R}^{k \times n} \to \mathbb{R}^n$ for each grammar rule $r = (A \to x(B_1, \dots, B_k)) \in R$, which maps the encodings of all children to an encoding of the parent node. Here, n is the encoding dimensionality. As such, the grammar guides our encoding and fixes the number and order of inputs for our encoding functions f^r. Note that, if $k = 0$, f^r is a constant.

Next, we apply the functions f^r recursively during parsing, yielding a vectorial encoding of the overall tree. More precisely, our encoder ϕ is defined by the recursive equation

$$\phi\Big(x(\hat{y}_1, \dots, \hat{y}_K), A\Big) = f^r(\phi(\hat{y}_1, B_1), \dots, \phi(\hat{y}_1, B_k)), \tag{3}$$

where $r = A \to x(B_1, \dots, B_k)$ is the first rule in the sequence that generates $x(\hat{y}_1, \dots, \hat{y}_K)$ from A. As initial nonterminal A, we use the grammar's starting symbol S. Refer to Algorithm 3 for details. Figure 2a–d shows an example of the scheme.

We implement f^r as a single-layer feedforward neural network of the form $f^r(\mathbf{y}_1, \dots, \mathbf{y}_k) = \tanh(\sum_{j=1}^{k} \boldsymbol{U}^{r,j} \cdot \mathbf{y}_j + \mathbf{a}^r)$, where the weight matrices $\boldsymbol{U}^{r,j} \in \mathbb{R}^{n \times n}$ and the bias vectors $\mathbf{a}^r \in \mathbb{R}^n$ are parameters to be learned. For optional and starred nonterminals,

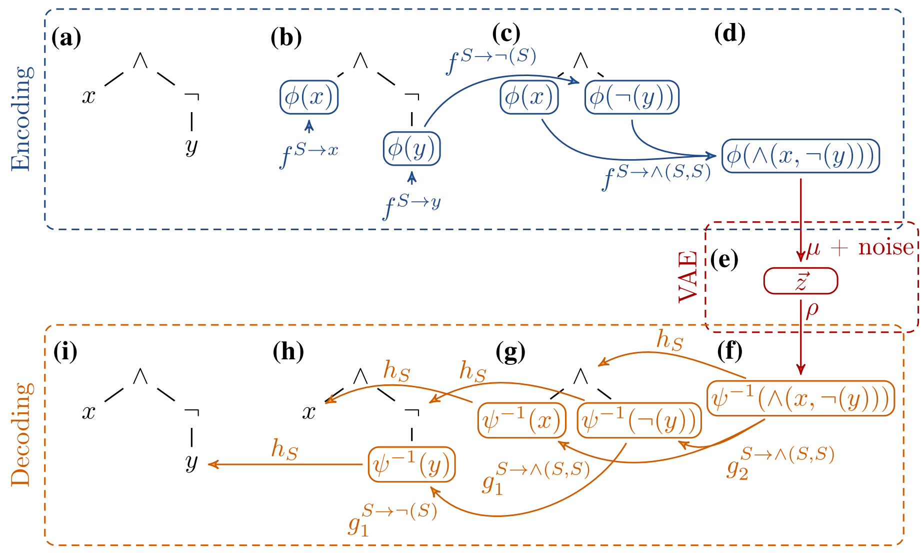

Fig. 2 An illustration of the recursive tree grammar autoencoder (RTG-AE) for the tree $\hat{x} = \wedge(x, \neg(y))$. Steps **a**–**d** encode the tree as the vector $\phi(\wedge(x, \neg(y)))$ (also refer to Algorithm 3). Step **e** maps it to the VAE latent space vector **z**. Steps **f**–**i** decode the vector back to the tree $\wedge(x, \neg(y))$ (also refer to Algorithm 4)

we further define $f^{B?\to\varepsilon} = f^{B^*\to\varepsilon} = \mathbf{0}$, $f^{B?\to B}(\mathbf{y}) = \mathbf{y}$, and $f^{B^*\to B,B^*}(\mathbf{y}_1, \mathbf{y}_2) = \mathbf{y}_1 + \mathbf{y}_2$. In other words, the empty string ε is encoded as the zero vector, an optional nonterminal is encoded via the identity, and starred nonterminals are encoded via a sum, following the recommendation of Xu et al. (2019) for graph neural nets.

Algorithm 3 Our encoding scheme, as first proposed in Paaßen et al. (2020), which extends bottom-up parsing for a deterministic regular tree grammar $\mathcal{G} = (\Phi, \Sigma, R, S)$. For each rule $r \in R$ we assume an encoding function f^r. The algorithm receives a tree as input and returns a nonterminal symbol, a rule sequence that generates the tree from that nonterminal symbol, and a vectorial encoding.

```
1:  function ENCODE(a tree $\hat{x} = x(\hat{y}_1, \ldots, \hat{y}_k)$)
2:      for $j \in \{1, \ldots, k\}$ do
3:          $B_j, \bar{r}_j, \vec{y}_j \leftarrow$ ENCODE($\hat{y}_j$).
4:      end for
5:      if $\exists A \in \Phi : A \to x(B_1, \ldots, B_k) \in R$ then
6:          $r \leftarrow \big(A \to x(B_1, \ldots, B_k)\big)$.
7:          return $A, (r, \bar{r}_1, \ldots, \bar{r}_k), f^r(\vec{y}_1, \ldots, \vec{y}_k)$.
8:      else
9:          Error; $\hat{x}$ is not in $\mathcal{L}(\mathcal{G})$.
10:     end if
11: end function
```

We can show that Algorithm 3 returns without error if and only if the input tree is part of the grammar's tree language.

Theorem 3 *Let $\mathcal{G} = (\Phi, \Sigma, R, S)$ be a deterministic regular tree grammar. Then, it holds: $\hat{x}$ is a tree in $\mathcal{L}(\mathcal{G})$ if and only if Algorithm 3 returns the nonterminal S as first output. Further, if Algorithm 3 returns with S as first output and some rule sequence $\bar{r}$ as second output, then $\bar{r}$ uniquely generates $\hat{x}$ from S. Finally, Algorithm 3 has $\mathcal{O}(|\hat{x}|)$ time and space complexity.*

Proof Refer to Appendix A.3. □

3.2 Decoder

Our decoder is a stochastic version of a given regular tree grammar $\mathcal{G} = (\Phi, \Sigma, R, S)$, controlled by two kinds of neural network. First, for any nonterminal $A \in \Phi$, let L_A be the number of rules in R with A on the left hand side. For each $A \in \Phi$, we introduce a linear layer $h_A : \mathbb{R}^n \to \mathbb{R}^{L_A}$ with $h_A(\mathbf{x}) = \boldsymbol{V}^A \cdot \mathbf{x} + \mathbf{b}^A$. To decode a tree from a vector $\mathbf{x} \in \mathbb{R}^n$ and a nonterminal $A \in \Phi$, we first compute rule scores $\boldsymbol{\lambda} = h_A(\mathbf{x})$ and then sample a rule $r_l = (A \to x(B_1, \ldots, B_k)) \in R$ from the softmax distribution $p_A(r_l|\mathbf{x}) = \exp(\lambda_l) / \sum_{l'=1}^{L_A} \exp(\lambda_{l'})$. Then, we apply the sampled rule and use a second kind of neural network to decide the vectorial encodings for each generated child nonterminal $B_1, \ldots, B_k$. In particular, for each grammar rule $r = (A \to x(B_1, \ldots, B_k)) \in R$, we introduce k feedforward layers $g_1^r, \ldots, g_k^r : \mathbb{R}^n \to \mathbb{R}^n$ and decode the vector representing the jth child as $\mathbf{y}_j = g_j^r(\mathbf{x})$. Finally, we decode the children recursively until no nonterminal is left. More precisely, the tree decoding is guided by the recursive equation

$$\psi(\mathbf{x}, A) = x\Big(\psi(\mathbf{y}_1, B_1), \ldots, \psi(\mathbf{y}_k, B_k)\Big), \tag{4}$$

where the rule $r = x \to A(B_1, \ldots, B_k)$ is sampled from p_A as specified above and $\mathbf{y}_j = g_j^r(\mathbf{x})$. As initial nonterminal argument we use the grammar's starting symbol S. For details, refer to Algorithm 4. Figure 2f–i shows an example of the scheme. Note that the time and space complexity is $\mathcal{O}(|\hat{x}|)$ for output tree $\hat{x}$ because each recursion step adds exactly one terminal symbol. Since the entire tree needs to be stored, the space complexity is also $\mathcal{O}(|\hat{x}|)$. Also note that Algorithm 4 is not generally guaranteed to halt (Chi, 1999). In practice, we solve this problem by imposing a maximum number of generated rules.

Algorithm 4 Our decoding scheme, similar to the scheme in Paaßen et al. (2020), which samples a rule sequence from a regular tree grammar $\mathcal{G} = (\Phi, \Sigma, R, S)$. The algorithm receives a vector $\vec{x}$ and the starting nonterminal S as input and then samples rules with probabilities determined by functions h_A for each nonterminal $A \in \Phi$ and decoding functions g^r_j for each rule $r = A \to x(B_1, \dots B_k) \in R$.

```
1: function DECODE(vector $\vec{x} \in \mathbb{R}^n$, nonterminal $A \in \Phi$)
2:     Let $r_1, \dots, r_{L_A} \in R$ be all rules with $A$ on the left hand side.
3:     Compute softmax weights $\vec{\lambda} \leftarrow h_A(\vec{x})$.
4:     Sample $r_l$ with probability
5:         $p_A(r_l|\vec{x}) = \exp(\lambda_l) / \sum_{l'=1}^{L_A} \exp(\lambda_{l'})$.
6:     Let $r_l = A \to x(B_1, \dots, B_k)$.
7:     for $j \in \{1, \dots, k\}$ do
8:         $\vec{y}_j \leftarrow g^{r_l}_j(\vec{x})$.
9:         $\hat{y}_j \leftarrow$ DECODE($\vec{y}_j$, $B_j$).
10:        $\vec{x} \leftarrow \vec{x} - \vec{y}_j$.
11:    end for
12:    return $x(\hat{y}_1, \dots, \hat{y}_k)$.
13: end function
```

For optional nonterminals, we introduce a classifier $h_{B?}$ which decides whether to apply $B? \to B$ or $B? \to \varepsilon$, and we define $g_1^{B? \to B}(\mathbf{y}) = \mathbf{y}$. For starred nonterminals, we introduce h_{B*} which decides whether to apply $B^* \to B, B^*$ or $B^* \to \varepsilon$, and we introduce new decoding layers $g_1^{B^* \to B, B^*}$ and $g_2^{B^* \to B, B^*}$.

An interesting special case are trees that implement lists. For example, consider a carbon chain CCC from chemistry. In the SMILES grammar (Weininger, 1988), this is represented as a binary tree of the form single_chain(single_chain(single_chain(chain_end, C), C), C), i.e. the symbol 'single_chain' acts as a list operator. In such a case, we recommend to use a recurrent neural network to implement the decoding function $g_1^{\text{Chain} \to \text{single\_chain(Chain,Atom)}}$, such as a gated recurrent unit (GRU) (Cho et al., 2014). In all other cases, we stick with a simple feedforward layer. We consider this issue in more detail in Appendix D.

Algorithm 5 A scheme to compute the variational autoencoder (VAE) loss from Equation 2 for our approach. The algorithm receives a tree $\hat{x}$ as input and returns the VAE loss, according to a mean function μ, a standard deviation function σ, a VAE decoding layer ρ, decoding functions $g^r_j : \mathbb{R}^n \to \mathbb{R}^n$, and VAE hyperparameters $s, \beta \in \mathbb{R}^+$.

```
1:  function LOSS(a tree $\hat{x}$)
2:      $A, (r_1, \ldots, r_T), \vec{x} \leftarrow$ ENCODE($\hat{x}$) via Algorithm 3.
3:      $\vec{\mu} \leftarrow \mu(\vec{x})$.    $\vec{\sigma} \leftarrow \sigma(\vec{x})$.    $\vec{\epsilon} \sim \mathcal{N}(\vec{0}|s \cdot \boldsymbol{I}^{n_{\text{VAE}}})$.
4:      $\vec{z} \leftarrow \vec{\mu} + \vec{\epsilon} \odot \vec{\sigma}$.
5:      Initialize a stack $\mathcal{S}$ with $\rho(\vec{z})$ on top.
6:      Initialize $\ell \leftarrow \beta \cdot \big(\sum_{j=1}^{n_{\text{VAE}}} \mu_j^2 + \sigma_j^2 - \log[\sigma_j^2] - 1\big)$.
7:      for $t \leftarrow 1, \ldots, T$ do
8:          Let $r_t = A \to x(B_1, \ldots, B_k)$.
9:          Pop $\vec{x}_t$ from the top of $\mathcal{S}$.
10:         Compute $p_A(r_t|\vec{x}_t)$ as in line 5 of Algorithm 4.
11:         $\ell \leftarrow \ell - \log\big[p_A(r_t|\vec{x}_t)\big]$.
12:         for $j \leftarrow 1, \ldots, k$ do
13:             $\vec{y}_j \leftarrow g_j^{r_t}(\vec{x}_t)$.
14:             $\vec{x}_t \leftarrow \vec{x}_t - \vec{y}_j$.
15:         end for
16:         Push $\vec{y}_k, \ldots, \vec{y}_1$ onto $\mathcal{S}$.
17:     end for
18:     return $\ell$.
19: end function
```

3.3 Training

We train our recursive tree grammar autoencoder (RTG-AE) in the variational autoencoder (VAE) framework, i.e. we try to minimize the loss in Eq. 2. More precisely, we define the encoding probability density $q_\phi(\mathbf{z}|\hat{x})$ as the Gaussian with mean $\mu(\phi(\hat{x}))$ and covariance matrix $\text{diag}\big[\sigma(\phi(\hat{x}))\big]$, where the functions $\mu : \mathbb{R}^n \to \mathbb{R}^{n_{\text{VAE}}}$ and $\sigma : \mathbb{R}^n \to \mathbb{R}^{n_{\text{VAE}}}$ are defined as

$$\begin{aligned}\mu(\mathbf{x}) &= \boldsymbol{U}^\mu \cdot \mathbf{x} + \mathbf{a}^\mu,\\ \sigma(\mathbf{x}) &= \exp\Big(\frac{1}{2} \cdot [\boldsymbol{U}^\sigma \cdot \mathbf{x} + \mathbf{a}^\sigma]\Big),\end{aligned} \tag{5}$$

where $\boldsymbol{U}^\mu, \boldsymbol{U}^\sigma \in \mathbb{R}^{n_{\text{VAE}} \times n}$ and $\mathbf{a}^\mu, \mathbf{a}^\sigma \in \mathbb{R}^{n_{\text{VAE}}}$ are additional parameters.

To decode, we first transform the encoding vector $\mathbf{z}$ with a single feedforward layer $\rho : \mathbb{R}^{n_{\text{VAE}}} \to \mathbb{R}^n$ and then apply the decoding scheme from Algorithm 4. As the decoding probability $p_\psi(\hat{x}|\mathbf{z})$, we use the product over all probabilities $p_A(r_t|\mathbf{x}_t)$ from line 5 of Algorithm 4, i.e. the probability of always choosing the correct grammar rule during decoding, provided that all previous choices have already been correct. The negative logarithm of this product can also be interpreted as the crossentropy loss between the correct rule

Table 1 The statistics for all four dataset

	Boolean	Expressions	SMILES	Pysort
Dataset size	34,884	104,832	249,456	294
Avg. tree size	5.21	8.95	83.66	57.97
Avg. depth	3.18	4.32	21.29	9.25
Max. rule seq. length	14	11	285	168
No. symbols	5	9	43	54
No. grammar rules	5	9	46	54

sequence and the softmax probabilities from line 5 of Algorithm 4. The details of our loss computation are given in Algorithm 5. Note that the time and space complexity is $\mathcal{O}(|\hat{x}|)$ because the outer loop from line 7–17 runs $T = |\hat{x}|$ times, and the inner loop in lines 12–15 runs $|\hat{x}| - 1$ times in total because every node takes the role of child exactly once (except for the root). Because the loss is differentiable, we can optimize it using gradient descent schemes such as Adam (Kingma & Ba, 2015). The gradient computation is performed by the pyTorch autograd system (Paszke et al., 2019).

4 Experiments and discussion

We evaluate the performance of RTG-AEs on four datasets, namely:

Boolean Randomly sampled Boolean formulae over the variables x and y with at most three binary operators, e.g. $x \wedge \neg y$ or $x \wedge y \wedge (x \vee \neg y)$.
Expressions Randomly sampled algebraic expressions over the variable x of the form $3 * x + \sin(x) + \exp(2/x)$, i.e. consisting of a binary operator plus a unary operator plus a unary of a binary. This dataset is taken from (Kusner et al., 2017).
SMILES Roughly 250k chemical molecules as SMILES strings (Weininger, 1988), as selected by (Kusner et al., 2017).
Pysort 29 Python sorting programs and manually generated preliminary development stages of these programs, resulting in 294 programs overall.
Table 1 shows dataset statistics, Appendix C lists the grammars for each of the datasets as well as the detailed sampling strategies for Boolean and Expressions.
We compare RTG-AEs to the following baselines.
GVAE Grammar variational autoencoders (Kusner et al., 2017) are grammar-based auto-encoders for strings. A string is first parsed by a context-free grammar, yielding a sequence of grammar rules. Next, the rule sequence is encoded via one-hot-coding, followed by three layers of 1D convolutions, followed by a fully connected layer. Note that this requires the maximum sequence length to be fixed in advance. Decoding occurs via a three-layer gated recurrent unit (Cho et al., 2014, GRU), followed by masking out invalid rule sequences according to the grammar. In this paper, we slightly adapt GVAE because we apply it to rule sequences of a regular tree grammar instead of a context-free grammar (otherwise, GVAE could not be applied to trees). Note that GVAE is different from RTG-AE because it uses sequential processing instead of recursive processing.
GRU-TG-AE Even though GVAE uses sequential instead of recursive processing, it is not a strict ablation of RTG-AE because it uses different architectures for encoding (conv-nets) and decoding (GRUs). Therefore, we also introduce an ablation of RTG-AE

Table 2 The number of parameters for all models on all datasets

Model	Boolean	Expressions	Smiles	Pysort
D-VAE	249,624	252,828	1,979,038	1,859,417
GVAE	198,316	197,536	1,928,431	1,630,263
TES-AE	500	900	11,776	13,824
GRU-TG-AE	67,221	70,025	437,080	445,280
RTG-AE	104,021	165,125	5,161,816	5,493,600

which uses GRUs for both encoding and decoding, but otherwise the same architecture. We call this baseline GRU-TG-AE.

D-VAE Directed acyclic graph variational autoencoders (Zhang et al., 2019) encode an input directed acyclic graph via a graph neural net that computes encodings following the structure of the graph—and hence becomes equivalent to a recursive neural net, as it is used by RTG-AE. However, the encoding does not use any grammatical information. It simply encodes the symbol at each node via one-hot coding and uses it as an additional input of the recursive net. For decoding, D-VAE uses the following recurrent scheme: Update the graph representation $\mathbf{h}$ with a GRU. Sample a new node from a softmax distribution over node types, where the scores for each type are computed via an MLP from $\mathbf{h}$. Then, for each previously sampled node, compute an edge probability via an MLP based on the encoding of the current node and the encoding of the previous node and sample the edge from a Bernoulli distribution with the computed probability. Continue with sampling nodes and edges until a special end token is sampled. Note that this scheme is different from RTG-AE because it does not use grammar knowledge. So D-VAE serves as the ablation of our model with recursive processing, learned encoding, but grammar-free processing.

TES-AE Tree Echo State Auto Encoders (Paaßen et al., 2020) are a predecessor to RTG-AE. TES-AEs use the same, recursive, grammar-based encoding and decoding scheme as presented in Sect. 3. However, the training scheme is entirely different: TES-AEs set the parameters for encoding layers f^r and decoding layers g^r_j randomly. Only the rule classifiers h_A are trained. This implies that the encoding for each tree is generated by an untrained function, as usual in an echo state network paradigm. Accordingly, the model needs to use a very high-dimensional encoding space to make sure that the untrained encodings still retain sufficient information to represent the tree. On the upside, treating f^r and g^r_j as fixed reduces the trainable parameters massively, and speeds up the training. TES-AEs serve as an ablation of RTG-AE because it uses recursive processing and grammar knowledge but a random encoding.

The number of parameters for all models on all datasets is shown in Table 2. We trained all neural networks using Adam (Kingma & Ba, 2015) with a learning rate of 10^{-3} and a `ReduceLROnPlateau` scheduler with minimum learning rate 10^{-4}, following the learning procedure of Kusner et al. (2017) as well as Dai et al. (2018). We sampled 100k trees for the first two and 10k trees for the latter two datasets. To obtain statistics, we repeated the training ten times with different samples (by generating new data for boolean and expressions and by doing a 10-fold crossvalidation for SMILES and pysort). Following

Table 3 The average autoencoding RMSE (± SD)

model	Boolean	Expressions	SMILES	Pysort
D-VAE	4.32 ± 0.41	5.84 ± 0.32	132.70 ± 57.02	70.67 ± 7.20
GVAE	3.61 ± 0.51	5.84 ± 2.00	594.92 ± 5.99	1.44 ± 5.18
TES-AE	2.62 ± 0.26	2.09 ± 0.18	581.08 ± 25.99	**20.41** ± **4.27**
GRU-TG-AE	1.98 ± 0.53	3.70 ± 0.31	482.88 ± 129.52	9363.32 ± 1.44
RTG-AE	**0.83** ± **0.23**	**0.77** ± **0.23**	**111.14** ± **159.97**	38.14 ± 3.63

Zhang et al. (2019), we used a batch size of 32 for all approaches.[1] For each approach and each dataset, we optimized the regularization strength β and the sampling standard deviation s in a random search with 20 trials over the range $[10^{-5}, 1]$, using separate data. For the first two datasets, we set $n = 100$ and $n_{\text{VAE}} = 8$, whereas for the latter two we set $n = 256$ and $n_{\text{VAE}} = 16$. For TES-AE, we followed the protocol of Paaßen et al. (2020), training on a random subset of 500 training data points, and we optimized the sparsity, spectral radius, regularization strength with the same hyper-parameter optimization scheme.

The SMILES experiment was performed on a computation server with a 24core CPU and 48GB RAM, whereas all other experiments were performed on a consumer grade laptop with Intel i7 4core CPU and 16GB RAM. All experimental code, including all grammars and implementations, is available at https://gitlab.com/bpaassen/RTGAE.

We measure autoencoding error on test data in terms of the root mean square tree edit distance (Zhang & Shasha, 1989). We use the root mean square error (RMSE) rather than log likelihood because D-VAE measures log likelihood different to GVAE, GRU-TG-AE, and RTG-AE, and TES-AE does not measure log likelihood at all. By contrast, the RMSE is agnostic to the underlying model. Further, we use the tree edit distance as a tree metric because it is defined on all possible labeled trees without regard for the underlying distribution or grammars (Bille, 2005) and hence does not favor any of the model.

The RMSE results are shown in Table 3. We observe that RTG-AE achieves significantly lower errors compared to all baselines on the first two datasets ($p < 0.001$ in a Wilcoxon signed rank test), significantly lower than all but D-VAE on the SMILES dataset ($p < 0.01$), and significantly lower than all but TES-AE on the Pysort dataset ($p < 0.001$). On SMILES, we note that we used a GRU in RTG-AE to represent chains of atoms, as described in Sect. 3. The vanilla RTG-AE performs considerably worse with an RMSE of 594.92 (the same as GVAE). On Pysort, TES-AE performs best, which is likely due to the fact that Pysort is roughly 100 times smaller than the other datasets while also having the most grammar rules. For this dataset, the dataset size was likely too small to fit a deep model.

Training times in seconds are shown in Table 4. We note that TES-AE is always the fastest because it only needs to fit the last layer. All other methods use deep learning. Among these models, RTG-AE had the lowest training times, likely because it consists of feedforward layers (except for one GRU layer in the SMILES case) whereas all other models use GRUs.

[1] On the Pysort dataset, the batch size of D-VAE had to be reduced to 16 to avoid memory overload.

Table 4 The average runtime in seconds (± SD) as measured by Python

model	Boolean	Expressions	SMILES	Pysort
D-VAE	882.0 ± 45.9	1201.5.0 ± 24.8	64225.9 ± 4500.3	66789.9 ± 3555.2
GVAE	1316.5 ± 86.7	1218.8 ± 73.8	7771.2 ± 1524.2	1393.7 ± 120.9
TES-AE	**0.9** ± **0.1**	**1.5** ± **0.2**	**583.6** ± **103.6**	**10.1** ± **0.6**
GRU-TG-AE	440.8 ± 30.4	734.5 ± 50.8	334.2 ± 42.7	648.6 ± 50.1
RTG-AE	221.5 ± 4.7	357.4 ± 13.2	1903.4 ± 34.1	399.9 ± 21.9

Table 5 The rate of syntactically correct trees decoded from standard normal random vectors

Model	Boolean (%)	Expressions (%)	SMILES (%)	Pysort (%)
D-VAE	28.8	23.8	0.1	1.4
GVAE	90.4	99.8	0	99.9
GRU-TG-AE	99.9	98.2	5.9	0
RTG-AE	98.5	99.6	37.3	99.7

To evaluate the ability of all models to generate syntactically valid trees, we sampled 1000 standard normal random vectors and decoded them with all models.[2] Then, we checked the syntactic correctness syntax with the regular tree grammar of the respective dataset (refer to Appendix C). The percentage of correct trees is shown in Table 5. Unsurprisingly, D-VAE has the worst results across the board because it does not use grammatical knowledge for decoding. In principle, the other models should always have 100% because their architecture guarantees syntactic correctness. Instead, we observe that the rates drop far below 100% for GRU-TG-AE on Pysort and for all models on SMILES. This is because the decoding process can fail if it gets stuck in a loop. Overall, RTG-AE, fails the least with an average of 83.74% across datasets, whereas D-VAE has 13.53%, GVAE has 72.53%, and GRU-TG-AE has 51%.

We also evaluated the utility of the autoencoder for optimization, in line with Kusner et al. (2017). The idea of these experiments is to find an optimal tree according to some objective function by using a gradient-free optimizer, such as Bayesian optimization, in the latent space of our tree autoencoder. If the optimizer is able to achieve a better objective function value, this indicates that the latent space behaves smoothly with respect to the objective function and, thus, may be a useful representation of the trees.

Kusner et al. (2017) considered two data sets, namely Expressions and SMILES. For the Expressions dataset, Kusner et al. (2017) suggest as objective function the log mean square error compared to the ground truth expression $\frac{1}{3} + x + \sin(x * x)$ for 1000 linearly spaced values of x in the range $[-10, +10]$. So any expression which produces similar outputs as the ground truth will achieve a good objective function value, but expressions that behave very differently achieve a worse error. For the SMILES dataset, Kusner et al. (2017) suggest an objective function which includes a range of chemically relevant properties, such

[2] We excluded TES-AE in this analysis because it does not guarantee a Gaussian distribution in the latent space and, hence, is not compatible with this sampling approach.

Table 6 The optimization scores for the Expressions (lower is better) and SMILES (higher is better) with median tree and median score ($\pm\frac{1}{2}$ IQR)

Model	Median tree	Median score
	Expressions	
D-VAE	x	0.49 ± 0.00
GVAE	x + 1 + sin(3 + 3)	0.46 ± 0.00
TES-AE	x	0.49 ± 0.00
GRU-TG-AE	x	0.49 ± 0.00
RTG-AE	x + 1 + sin(x * x)	$\mathbf{0.33 \pm 0.05}$
	SMILES	
D-VAE	n.a.	n.a.
GVAE	n.a.	n.a.
TES-AE	CCO	-0.31 ± 0.50
GRU-TG-AE	C=C	-2.23 ± 1.56
RTG-AE	CCCCCCC	$\mathbf{2.57 \pm 0.31}$

as logP, synthetic availability, and cycle length. Here, higher values are better. For details, please refer to the original paper and/or our source code at https://gitlab.com/bpaassen/RTG-AE. We used both objective functions exactly as specified in the original work. For SMILES, we also re-trained all models with $n_{\text{VAE}} = 56$ and 50k training samples to be consistent with Kusner et al. (2017).

We tried to use the same optimizer as Kusner et al. (2017), namely Bayesian Optimization, but were unable to get the original implementation to run. Instead, we opted for a CMA-ES optimizer, namely the `cma` implementation in Python. CMA-ES is a well-established method for high-dimensional, gradient-free otpimization and has shown competitive results to Bayesian optimization in some cases (Loshchilov & Hutter, 2016). Conceptually, CMA-ES fits well with variational autoencoders because both VAEs and CMA-ES use Gaussian distributions in the latent space. In particular, we initialize CMA-ES with the standard Gaussian and then let it adapt the mean and covariance matrix to move to samples with better objective function value. As hyper-parameters, we set 15 iterations and a budget of 750 objective function evaluations, which is the same as Kusner et al. (2017). To obtain statistics, we performed the optimization 10 times for each autoencoder and each dataset.

The median results (± inter-quartile ranges) are shown in Table 6. We observe that RTG-AEs significantly outperform all baselines on both datasets ($p < 0.01$ in a Wilcoxon rank-sum test) by a difference of several inter-quartile ranges. On the SMILES dataset, D-VAE failed with an out-of-memory error during re-training and G-VAE failed because CMA-ES could not find any semantically valid molecule in the latent space, such that both methods receive an n.a. score. We note that, on both datasets, our results for GVAE are worse than the ones reported by Kusner et al. (2017), which is likely because Bayesian optimization is a stronger optimizer in these cases. Still, our results show that even the weaker CMA-ES optimizer can consistently achieve good scores in the RTG-AE latent space. We believe there are two reasons for this: First, RTG-AE tends to have a higher rate of syntactically correct trees (refer to Table 5); second, recursive processing tends to cluster similar trees together (Paaßen et al., 2020; Tiňo & Hammer, 2003) in a fractal fashion, such that an optimizer only needs to find a viable cluster and optimize within it. Figure 3 shows a t-SNE visualization of the latent spaces, indicating a cluster structure for TES-AE and RTG-AE, whereas GVAE yields a single blob. We also performed a 30-means clustering in the latent

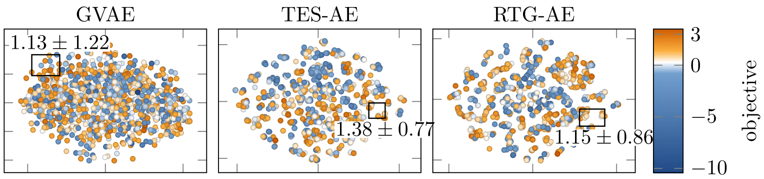

Fig. 3 A 2D t-SNE reduction of the codes of 1000 random molecules from the SMILES dataset by GVAE (left), TES-AE (center), and RTG-AE (right). Color indicates the objective function value. Rectangles indicate the 30-means cluster with highest mean objective function value ± std

space, revealing that the cluster with highest objective function value for TES-AE and RTG-AE had higher mean and lower variance compared to the one for GVAE (black rectangles in Fig. 3).

5 Conclusion

In this contribution, we introduced the recursive tree grammar autoencoder (RTG-AE), a novel neural network architecture that combines variational autoencoders with recursive neural networks and regular tree grammars. In particular, our approach encodes a tree with a bottom-up parser, and decodes it with a tree grammar, both learned via neural networks and variational autoencoding. Experimentally, we showed that the unique combination of recursive processing, grammatical knowledge, and deep learning generally improves autoencoding error, training time, and optimization performance beyond existing models that use only two of these features, but not all three. The lower autoencoding error can be explained by three conceptual observations: First, recursive processing follows the tree structure whereas sequential processing introduces long-range dependencies between children and parents in the tree; second, grammatical knowledge avoids obvious decoding mistakes by limiting the terminal symbols we can choose; third, deep learning allows to adjust all model parameters to the data instead of merely the last layer. The lower training time can be explained by the fact that RTG-AE uses (almost) only feedforward layers whereas other models require recurrent layers. The improved optimization performance is likely because recursive processing yields a clustered latent space that helps optimization, grammar knowledge avoids a lot of trees with low objective function values, and variational autoencoding encourages a smooth encoding space that makes it easier to optimize over the space.

Nonetheless, our work still has notable limitations that provide opportunities for future work. First, we need a pre-defined, deterministic regular tree grammar, which may not be available in all domains. Second, we only consider optimization as downstream task, whereas the utility of our model for other tasks—such as time series prediction—remains to be shown. Third, on the (small) Pysort dataset, RTG-AE was outperformed by the much simpler TES-AE model, indicating that a sufficient dataset size is necessary to fit RTG-AE. This data requirement ought to be investigated in more detail. Finally, RTG-AE integrates syntactic domain knowledge (in form of a grammar) but currently does not consider semantic constraints. Such constraints could be integrated in future extensions of the model.

Appendix A: Proofs

For the purpose of our tree language proofs, we first introduce a few auxiliary concepts. First, we extend the definition of a regular tree grammar slightly to permit multiple starting symbols. In particular, we re-define a regular tree grammar as a 4-tuple $\mathcal{G} = (\Phi, \Sigma, R, \mathcal{S})$, where Φ and Σ are disjoint finite sets and R is a rule set as before, but $\mathcal{S}$ is now a subset of Φ. Next, we define the partial tree language $\mathcal{L}(\mathcal{G}|A)$ for nonterminal $A \in \Phi$ as the set of all trees which can be generated from nonterminal A using rule sequences from R. Further, we define the n-restricted partial language $\mathcal{L}_n(\mathcal{G}|A)$ for nonterminal $A \in \Phi$ as the set $\mathcal{L}_n(\mathcal{G}|A) := \left\{\hat{x} \in \mathcal{L}(\mathcal{G}|A) \middle| |\hat{x}| \leq n\right\}$, i.e. it includes only the trees up to size n. We define the tree language $\mathcal{L}(\mathcal{G})$ of $\mathcal{G}$ as the union $\mathcal{L}(\mathcal{G}) := \bigcup_{S \in \mathcal{S}} \mathcal{L}(\mathcal{G}|S)$.

As a final preparation, we introduce a straightforward lemma regarding restricted partial tree languages.

Lemma 1 *Let $\mathcal{G} = (\Phi, \Sigma, R, \mathcal{S})$ be a regular tree grammar.*

Then, for all $A \in \Phi$, all $n \in \mathbb{N}$ and all trees $\hat{x}$ it holds: $\hat{x} = x(\hat{y}_1, \ldots, \hat{y}_k)$ is in $\mathcal{L}_n(\mathcal{G}|A)$ if and only if there exist nonterminals $B_1, \ldots, B_k \in \Phi$ such that $A \to x(B_1, \ldots, B_k) \in R$ and $\hat{y}_j \in \mathcal{L}_{n_j}(\mathcal{G}|B_j)$ for all $j \in \{1, \ldots, k\}$ with $n_1 + \ldots + n_k = n - 1$.

Proof If $\hat{x} \in \mathcal{L}_n(\mathcal{G}|A)$, then there exists a sequence of rules $r_1, \ldots, r_T \in R$ which generates $\hat{x}$ from A. The first rule in that sequence must be of the form $A \to x(B_1, \ldots, B_k)$, otherwise $\hat{x}$ would not have the shape $\hat{x} = x(\hat{y}_1, \ldots, \hat{y}_k)$. Further, $r_2, \ldots, r_T$ must consist of subsequences $\bar{r}_1, \ldots, \bar{r}_k = r_2, \ldots, r_T$ such that $\bar{r}_j$ generates $\hat{y}_j$ from B_j for all $j \in \{1, \ldots, k\}$, otherwise $\hat{x} \neq x(\hat{y}_1, \ldots, \hat{y}_k)$. However, that implies that $\hat{y}_j \in \mathcal{L}(\mathcal{G}|B_j)$ for all $j \in \{1, \ldots, k\}$. The length restriction to n_j follows because—per definition of the tree size—the sizes $|\hat{y}_j|$ must add up to $|\hat{x}| - 1$.

Conversely, if there exist nonterminals $B_1, \ldots, B_k \in \Phi$ such that $r = A \to x(B_1, \ldots, B_k) \in R$ and $\hat{y}_j \in \mathcal{L}_{n_j}(\mathcal{G}|B_j)$ for all $j \in \{1, \ldots, k\}$, then there must exist rule sequences $\bar{r}_1, \ldots, \bar{r}_k$ which generate $\hat{y}_j$ from B_j for all $j \in \{1, \ldots, k\}$. Accordingly, $r, \bar{r}_1, \ldots, \bar{r}_k$ generates $\hat{x}$ from A and, hence, $\hat{x} \in \mathcal{L}(\mathcal{G}|A)$. The length restriction follows because $|\hat{x}| = 1 + |\hat{y}_1| + \ldots + |\hat{y}_k| \leq 1 + n_1 + \ldots + n_k = n$. □

A.1 Deterministic grammars imply unique rule sequences

Recall that we define a regular tree grammar as *deterministic* if no two rules have the same right-hand-side. Our goal is to show that every deterministic regular tree grammar is unambiguous, in the sense that there always exists a unique rule sequence generating the tree. We first prove an auxiliary result.

Lemma 2 *Let $\mathcal{G} = (\Phi, \Sigma, R, S)$ be a deterministic regular tree grammar. Then, for any two $A \neq A' \in \Phi$, $\mathcal{L}(\mathcal{G}|A) \cap \mathcal{L}(\mathcal{G}|A') = \emptyset$.*

Proof We perform a proof via induction over the tree size. First, consider trees $\hat{x}$ with $|\hat{x}| = 1$, that is, $\hat{x} = x()$. If $\hat{x} \in \mathcal{L}(\mathcal{G}|A)$ for some $A \in \Phi$, the rule $A \to x()$ must be in R.

Because $\mathcal{G}$ is deterministic, there can exist no $A' \neq A \in \Phi$ with $A' \to x() \in R$, otherwise there would be two rules with the same right-hand-side. Accordingly, $\hat{x}$ lies in $\mathcal{L}(\mathcal{G}|A)$ for at most one $A \in \Phi$.

Now, assume that the claim holds for all trees up to size n and consider a tree $\hat{x}$ with $|\hat{x}| = n + 1$. Without loss of generality, let $\hat{x} = x(\hat{y}_1, \ldots, \hat{y}_k)$. If $\hat{x} \in \mathcal{L}(\mathcal{G}|A)$ for some $A \in \Phi$, a rule of the form $A \to x(B_1, \ldots, B_k)$ must be in R, such that for all $j \in \{1, \ldots, k\}$: $\hat{y}_j \in \mathcal{L}(\mathcal{G}|B_j)$. Now, note that $|\hat{y}_j| \leq n$. Accordingly, our induction hypothesis applies and there exists no other $B'_j \neq B_j$ such that $\hat{y}_j \in \mathcal{L}(\mathcal{G}|B'_j)$. Further, there can exist no $A' \neq A$ with $A' \to x(B_1, \ldots, B_k)$, otherwise there would be two rules with the same right-hand-side. Accordingly, $\hat{x}$ lies in $\mathcal{L}(\mathcal{G}|A)$ for at most one $A \in \Phi$. □

Now, we prove the desired result.

Theorem 4 *Let $\mathcal{G} = (\Phi, \Sigma, R, S)$ be a deterministic regular tree grammar. Then, for any $\hat{x} \in \mathcal{L}(\mathcal{G})$ there exists exactly one sequence of rules $r_1, \ldots, r_T \in R$ which generates $\hat{x}$ from S.*

Proof In fact, we will prove a more general result, namely that the claim holds for all $\hat{x} \in \bigcup_{A \in \Phi} \mathcal{L}(\mathcal{G}|A)$. We perform an induction over the tree size.

First, consider trees $\hat{x}$ with $|\hat{x}| = 1$, that is, $\hat{x} = x()$. Then, the only way to generate $\hat{x}$ is by applying a single rule of the form $A \to x()$. Because $\mathcal{G}$ is deterministic, only one such rule can exist. Therefore, the claim holds.

Now, assume that the claim holds for all trees up to size n and consider a tree $\hat{x}$ with $|\hat{x}| = n + 1$. Without loss of generality, let $\hat{x} = x(\hat{y}_1, \ldots, \hat{y}_k)$. If $\hat{x} \in \mathcal{L}(\mathcal{G}|A)$ for some $A \in \Phi$, then there must exist a rule of the form $A \to x(B_1, \ldots, B_k)$ in R, such that for all $j \in \{1, \ldots, k\}$: $\hat{y}_j \in \mathcal{L}(\mathcal{G}|B_j)$. Due to the previous lemma and our induction hypothesis we know that for each j, there exists a unique nonterminal B_j and rule sequence $\bar{r}_j$, such that $\bar{r}_j$ generates $\hat{y}_j$ from B_j. Further, because $\mathcal{G}$ is deterministic, there can exist no other $A' \neq A \in \Phi$, such that $A' \to x(B_1, \ldots, B_k)$ in R. Therefore, the rule $A \to x(B_1, \ldots, B_k)$ concatenated with the rule sequences $\bar{r}_1, \cdots, \bar{r}_k$ is the only way to generate $\hat{x}$, as claimed. □

A.2 All regular tree grammars can be made deterministic

Theorem 5 *Let $\mathcal{G} = (\Phi, \Sigma, R, \mathcal{S})$ be a regular tree grammar. Then, there exists a regular tree grammar $\mathcal{G}' = (\Phi', \Sigma, R', \mathcal{S}')$ which is deterministic and where $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{G}')$.*

Proof This proof is an adaptation of Theorem 1.1.9 of Comon et al. (2008) and can also be seen as an analogue to conversion from nondeterministic finite state machines to deterministic finite state machines.

In particular, we convert $\mathcal{G}$ to $\mathcal{G}'$ via the following procedure.

We first initialize Φ' and R' as empty sets.

Second, iterate over k, starting at zero up to the maximum number of children in a rule in R, and iterate over all right-hand-sides $x(B_1, \ldots, B_k)$ that occur in rules in R. Next, we collect all $A_1, \ldots, A_m \in \Phi$ such that $A_i \to x(B_1, \ldots, B_k) \in R$ and add the set $A' = \{A_1, \ldots, A_m\}$ to Φ'.

Third, we perform the same iteration again, but this time we add grammar rules $\{A_1, \dots, A_m\} \to x(B'_1, \dots, B'_k)$ to R' for all combinations $B'_1, \dots, B'_k \in \Phi'$ where $B_j \in B'_j$ for all $j \in \{1, \dots, k\}$.

Finally, we define $\mathcal{S}' = \{A' \in \Phi' | A' \cap \mathcal{S} \neq \emptyset\}$.

It is straightforward to see that $\mathcal{G}'$ is deterministic because all rules with the same right-hand-side get replaced with rules with a unique left-hand-side.

It remains to show that the generated tree languages are the same. To that end, we first prove an auxiliary claim, namely that for all $A \in \Phi$ it holds: A tree $\hat{x}$ is in $\mathcal{L}_n(\mathcal{G}|A)$ if and only if there exists an $A' \in \Phi'$ with $A \in A'$ and $\hat{x} \in \mathcal{L}_n(\mathcal{G}'|A')$.

We show this claim via induction over n. First, consider $n = 1$. If $\hat{x} \in \mathcal{L}_1(\mathcal{G}|A)$, $\hat{x}$ must have the shape $\hat{x} = x()$, otherwise $|\hat{x}| > 1$, and the rule $A \to x()$ must be in R. Accordingly, our procedure assures that there exists some $A' \in \Phi'$ with $A \in A'$ and $A' \to x() \in R'$. Hence, $\hat{x} \in \mathcal{L}_1(\mathcal{G}'|A')$.

Conversely, if $\hat{x} \in \mathcal{L}_1(\mathcal{G}'|A')$ for some $A' \in \Phi'$ then $\hat{x}$ must have the shape $\hat{x} = x()$, otherwise $|\hat{x}| > 1$, and the rule $A' \to x()$ must be in R'. However, this rule can only be in R' if for any $A \in A'$ the rule $A \to x()$ was in R. Accordingly, $\hat{x} \in \mathcal{L}_1(\mathcal{G}|A)$.

Now, consider the case $n > 1$ and let $\hat{x} = x(\hat{y}_1, \dots, \hat{y}_k)$. If $\hat{x} \in \mathcal{L}_n(\mathcal{G}|A)$, Lemma 1 tells us that there must exist nonterminals $B_1, \dots, B_k \in \Phi$ such that $A \to x(B_1, \dots, B_k) \in R$ and $\hat{y}_j \in \mathcal{L}_{n_j}(\mathcal{G}|B_j)$ for all $j \in \{1, \dots, k\}$ with $n_1 + \dots + n_k = n - 1$. Hence, by induction we know that there exist $B'_j \in \Phi'$ such that $B_j \in B'_j$ and $\hat{y}_j \in \mathcal{L}_{n_j}(\mathcal{G}'|B'_j)$ for all $j \in \{1, \dots, k\}$. Further, our procedure for generating $\mathcal{G}'$ ensures that there exists some $A' \in \Phi'$ such that the rule $A' \to x(B'_1, \dots, B'_k)$ is in R'. Hence, Lemma 1 tells us that $\hat{x} \in \mathcal{L}_n(\mathcal{G}'|A')$.

Conversely, if $\hat{x} \in \mathcal{L}_n(\mathcal{G}'|A')$ then Lemma 1 tells us that there must exist nonterminals $B'_1, \dots, B'_k \in \Phi'$ such that $A' \to x(B'_1, \dots, B'_k) \in R'$ and $\hat{y}_j \in \mathcal{L}_{n_j}(\mathcal{G}'|B'_j)$ for all $j \in \{1, \dots, k\}$ with $n_1 + \dots + n_k = n - 1$. Hence, by induction we know that for all $j \in \{1, \dots, k\}$ and all $B_j \in B'_j$ we obtain $\hat{y}_j \in \mathcal{L}_{n_j}(\mathcal{G}|B_j)$. Further, our procedure for generating $\mathcal{G}'$ ensures that for any $A \in A'$ there must exist a rule $A \to x(B_1, \dots, B_k) \in R$ for some $B_1, \dots, B_k \in \Phi$ and $B_j \in B'_j$ for some combination of $B'_j \in \Phi'$. Hence, Lemma 1 tells us that $\hat{x} \in \mathcal{L}_n(\mathcal{G}|A)$. This concludes the induction.

Now, note that $\hat{x} \in \mathcal{L}(\mathcal{G})$ implies that there exists some $S \in \mathcal{S}$ with $\hat{x} \in \mathcal{L}(\mathcal{G}|S)$. Our auxiliary result tells us that there then exists some $S' \in \Phi'$ with $S \in \Phi'$ and $\hat{x} \in \mathcal{L}(\mathcal{G}'|S')$. Further, since $S' \cup \mathcal{S}$ contains at least S, our construction of $\mathcal{G}'$ implies that $S' \in \mathcal{S}'$. Hence, $\hat{x} \in \mathcal{L}(\mathcal{G}')$.

Conversely, if $\hat{x} \in \mathcal{L}(\mathcal{G}')$, there must exist some $S' \in \mathcal{S}'$ with $\hat{x} \in \mathcal{L}(\mathcal{G}'|S')$. Since $S' \in \mathcal{S}'$, there must exist at least one $S \in S'$ such that $S \in \mathcal{S}$. Further, our auxiliary result tells us that $\hat{x} \in \mathcal{L}(\mathcal{G}|S)$. Accordingly, $\hat{x} \in \mathcal{L}(\mathcal{G})$. □

As an example, consider the following grammar $\mathcal{G} = (\Phi, \Sigma, R, \mathcal{S})$ for logical formulae in conjuctive normal form.

$$\begin{aligned}
\Phi &= \{F, C, L, A\} \\
\Sigma &= \{\wedge, \vee, \neg, x, y\} \\
R &= \{F \to \wedge(C, F) | \vee (L, C) | \neg(A) | x | y, \\
&\quad C \to \vee(L, C) | \neg(A) | x | y, \\
&\quad L \to \neg(A) | x | y, \\
&\quad A \to x | y\} \\
\mathcal{S} &= \{F\}
\end{aligned}$$

The deterministic version of this grammar is $\mathcal{G}' = (\Phi', \Sigma, R', \mathcal{S}')$ with $\mathcal{S}' = \Phi'$ and

$$\begin{aligned}
\Phi' &= \{\{F, C, L, A\}, \{F, C, L\}, \{F, C\}, \{F\}\} \\
R' &= \Big\{ \{F, C, L, A\} \to x | y, \\
&\quad \{F, C, L\} \to \neg(\{F, C, L, A\}), \\
&\quad \{F, C\} \to \vee(\{F, C, L, A\}, \{F, C, L, A\}), \\
&\quad \{F, C\} \to \vee(\{F, C, L, A\}, \{F, C, L\}), \\
&\quad \{F, C\} \to \vee(\{F, C, L, A\}, \{F, C\}), \\
&\quad \{F, C\} \to \vee(\{F, C, L\}, \{F, C, L, A\}), \\
&\quad \{F, C\} \to \vee(\{F, C, L\}, \{F, C, L\}), \\
&\quad \{F, C\} \to \vee(\{F, C, L\}, \{F, C\}), \\
&\quad \{F\} \to \wedge(\{F, C, L, A\}, \{F, C, L, A\}), \\
&\quad \{F\} \to \wedge(\{F, C, L, A\}, \{F, C, L\}), \\
&\quad \{F\} \to \wedge(\{F, C, L, A\}, \{F, C\}), \\
&\quad \{F\} \to \wedge(\{F, C, L, A\}, \{F\}), \\
&\quad \{F\} \to \wedge(\{F, C, L\}, \{F, C, L, A\}), \\
&\quad \{F\} \to \wedge(\{F, C, L\}, \{F, C, L\}), \\
&\quad \{F\} \to \wedge(\{F, C, L\}, \{F, C\}), \\
&\quad \{F\} \to \wedge(\{F, C, L\}, \{F\}), \\
&\quad \{F\} \to \wedge(\{F, C\}, \{F, C, L, A\}), \\
&\quad \{F\} \to \wedge(\{F, C\}, \{F, C, L\}), \\
&\quad \{F\} \to \wedge(\{F, C\}, \{F, C\}), \\
&\quad \{F\} \to \wedge(\{F, C\}, \{F\}) \Big\}.
\end{aligned}$$

A.3 Proof of the Parsing Theorem

We first generalize the statement of the theorem to also work for regular tree grammars with multiple starting symbols. In particular, we show the following:

Theorem 6 *Let $\mathcal{G} = (\Phi, \Sigma, R, \mathcal{S})$ be a deterministic regular tree grammar. Then, it holds: $\hat{x}$ is a tree in $\mathcal{L}(\mathcal{G})$ if and only if Algorithm* 3 *returns a nonterminal $S \in \mathcal{S}$, a unique rule* *sequence $r_1, \dots, r_T$ that generates $\hat{x}$, and a vector $\phi(\hat{x}) \in \mathbb{R}^n$ for the input $\hat{x}$. Further, Algorithm 3 has $\mathcal{O}(|\hat{x}|)$ time and space complexity.*

Proof We structure the proof in three parts. First, we show that $\hat{x} \in \mathcal{L}(\mathcal{G})$ implies that Algorithm 3 returns with a nonterminal $S \in \mathcal{S}$, a unique generating rule sequence, and some vector $\phi(\mathbf{x}) \in \mathbb{R}^n$. Second, we show that if Algorithm 3 returns with a nonterminal $S \in \mathcal{S}$ and some rule sequence as well as some vector, then the tree generated by the rule sequence lies in $\mathcal{L}(\mathcal{G})$. Finally, we prove the complexity claim.

We begin by a generalization of the first claim. For all $A \in \Phi$ and for all $\hat{x} \in \mathcal{L}(\mathcal{G}|A)$ it holds: Algorithm 3 returns the nonterminal A, a rule sequence $r_1, \dots, r_T \in \mathbb{R}$ that generates $\hat{x}$ from A, and a vector $\phi(\hat{x})$. The uniqueness of the rule sequence $r_1, \dots, r_T$ follows from Theorem 4.

Our proof works via induction over the tree size. Let $A \in \phi$ be any nonterminal and let $\hat{x} \in \mathcal{L}_1(\mathcal{G}|A)$. Then, because $|\hat{x}| = 1$, $\hat{x}$ must have the form $\hat{x} = x()$ for some $x \in \Sigma$ and there must exist a (unique) rule of the form $r = A \to x() \in R$ for some $A \in \Phi$, otherwise $\hat{x}$ would not be in $\mathcal{L}_1(\mathcal{G}|A)$. Now, inspect Algorithm 3 for $\hat{x}$. Because $k = 0$, lines 2-4 do not get executed. Further, because $\mathcal{G}$ is deterministic, the $\exists$ Quantifier in line 5 is unique, such that r in line 6 is exactly $A \to x()$. Accordingly, line 7 returns exactly A, r, f^r.

Now, assume that the claim holds for all trees $\hat{y} \in \bigcup_{A \in \Phi} \mathcal{L}_n(\mathcal{G}|A)$ with $n \geq 1$. Then, consider some nonterminal $A \in \Phi$ and some tree $\hat{x} \in \mathcal{L}_{n+1}(\mathcal{G}|A)$ with size $|\hat{x}| = n + 1$. Because $|\hat{x}| > 1$, $\hat{x}$ must have the shape $\hat{x} = x(\hat{y}_1, \dots, \hat{y}_k)$ with $k > 0$ for some $x \in \Sigma$ and some trees $\hat{y}_1, \dots, \hat{y}_k$. Lemma 1 now tells us that there must exist nonterminals $B_1, \dots, B_k \in \Phi$ with $\hat{y}_j \in \mathcal{L}_{n_j}(\mathcal{G}|B_j)$ for all $j \in \{1, \dots, k\}$ such that $n_1 + \dots + n_k = n - 1$. Hence, our induction applies to all trees $\hat{y}_1, \dots, \hat{y}_k$.

Now, inspect Algorithm 3 once more. Due to induction we know that line 3 returns for each tree $\hat{y}_j$ a unique combination of nonterminal B_j and rule sequence $\bar{r}_j$, such that $\bar{r}_j$ generates $\hat{y}_j$ from B_j. Further, the rule $A \to x(B_1, \dots, B_k)$ must exist in R, otherwise $\hat{x} \notin \mathcal{L}_{n+1}(\mathcal{G}|A)$. Also, there can not exist another nonterminal $B \in \Phi$ with $B \to x(B_1, \dots, B_k) \in R$, otherwise $\mathcal{G}$ would not be deterministic. Therefore, r in line 6 is exactly $A \to x(B_1, \dots, B_k)$. It remains to show that the returned rule sequence does generate $\hat{x}$ from A. The first step in our generation is to pop A from the stack, replace A with $x(B_1, \dots, B_k)$, and push $B_k, \dots, B_1$ onto the stack. Next, B_1 will be popped from the stack and the next rules will be $\bar{r}_1$. Due to induction we know that $\bar{r}_1$ generates $\hat{y}_1$ from B_1, resulting in the intermediate tree $x(\hat{y}_1, B_2, \dots, B_k)$ and the stack $B_k, \dots, B_2$. Repeating this argument with the remaining rule sequence $\bar{r}_2, \dots, \bar{r}_k$ yields exactly $\hat{x}$ and an empty stack, which means that the rule sequence does indeed generate $\hat{x}$ from A.

For the second part, we also consider a generalized claim. In particular, if Algorithm 3 for some input tree $\hat{x}$ returns with some nonterminal A and some rule sequence $\bar{r}$, then $\bar{r}$ generates $\hat{x}$ from A. We again prove this via induction over the length of the input tree. If $|\hat{x}| = 1$, then $\hat{x} = x()$ for some x. Because $k = 0$, lines 2-4 of Algorithm 3 do not get executed. Further, there must exist some nonterminal $A \in \Phi$ such that $r = A \to x() \in R$, otherwise Algorithm 3 would return an error, which is a contradiction. Finally, the return values are A and r and and r generates $\hat{x} = x()$ from A as claimed.

Now, assume the claim holds for all trees with length up to $n \geq 1$ and consider a tree $\hat{x} = x(\hat{y}_1, \dots, \hat{y}_k)$ with $|\hat{x}| = n + 1$. Because $n + 1 > 1$, $k > 0$. Further, for all $j \in \{1, \dots, k\}$, $|\hat{y}_j| < |\hat{x}| = n + 1$, such that the induction hypothesis applies. Accordingly, line 3 returns nonterminals B_j and rule sequences $\bar{r}_j$, such that $\bar{r}_j$ generates $\hat{y}_j$ from B_j for all $j \in \{1, \dots, k\}$. Further, some nonterminal $A \in \Phi$ must exist such that $A \to x(B_1, \dots, B_k) \in R$, otherwise

Algorithm 3 would not return, which is a contradiction. Finally, the return values are A and $r, \bar{r}_1, \dots, \bar{r}_k$ and $r, \bar{r}_1, \dots, \bar{r}_k$ generates $\hat{x}$ from A using the same argument as above. This concludes the proof by induction.

Our actual claim now follows. In particular, if $\hat{x} \in \mathcal{L}(\mathcal{G})$, then there exists some $S \in \mathcal{S}$ such that $\hat{x} \in \mathcal{L}(\mathcal{G}|S)$. Accordingly, Algorithm 3 returns with S, a generating rule sequence from S, and a vector $\phi(\hat{x})$ as claimed. Conversely, if for some input tree $\hat{x}$ Algorithm 3 returns with some nonterminal $S \in \mathcal{S}$ and rule sequence $\bar{r}$, then $\bar{r}$ generates $\hat{x}$ from S, which implies that $\hat{x} \in \mathcal{L}(\mathcal{G})$.

Regarding time and space complexity, notice that each rule adds exactly one node tot he tree and that each recursion of Algorithm 3 adds exactly one rule. Accordingly, Algorithm 3 requires exactly $|\hat{x}|$ iterations. Since the returned rule sequence has length $|\hat{x}|$ and all other variables have constant size, the space complexity is the same. ☐

Appendix B: Determinism for optional and starred nonterminals

As mentioned in the main text, $A?$ denotes a nonterminal with the production rules $A? \to A$ and $A? \to \epsilon$, and $A*$ denotes a nonterminal with the production rules $A* \to A, A*$ and $A* \to \epsilon$, where the comma refers to concatenation. It is easy to see that these concepts can make a grammar ambiguous. For example, the grammar rule $A \to x(B*, B*)$ is ambiguous because for the partially parsed tree $x(B)$ we could not decide whether B has been produced by the first or second $B*$. Similarly, the two grammar rules $A \to x(B*, C)$ and $B \to x(C)$ are ambiguous because the tree $x(C)$ can be produced by both.

More generally, we define a regular tree grammar as deterministic if the following two conditions are fulfilled.

1. Any grammar rule $A \to x(B_1, \dots, B_k) \in R$ must be internally deterministic in the sense that any two neighboring symbols B_j, B_{j+1} must be either unequal or not both starred/optional.
2. Any two grammar rules $A \to x(B_1, \dots, B_k) \in R$ and $A' \to x(B'_1, \dots, B'_l) \in R$ must be non-intersecting, in the sense that the sequences $B_1, \dots, B_k$ and $B'_1, \dots, B'_l$, when interpreted as regular expressions, are not allowed to have intersecting languages.

Note that, if we do not have starred or optional nonterminals in any rule, the first condition is automatically fulfilled and the second condition collapses to our original definition of determinism, namely that no two rules are permitted to have the same right hand side. Further note that the intersection of regular expression languages can be checked efficiently via standard techniques from theoretical computer science.

If both conditions are fulfilled, we can adapt Algorithm 3 in a straightforward fashion by replacing line 5 with $\exists A \in \Phi, A \to x(B'_1, \dots, B'_l) \in R$ such that $B'_1, \dots, B'_l$ when interpreted as a regular expression matches $B_1, \dots, B_k$. There can be only one such rule, otherwise the second condition would not be fulfilled. We then the encodings $\mathbf{y}_1, \dots, \mathbf{y}_k$ to their matching nonterminals in the regular expression $B'_1, \dots, B'_l$ and proceed as before.

Appendix C: Grammars

In this section, we list the grammars for each of the datasets in the experiments.

Boolean The grammar is $\mathcal{G}_{\text{Bool}} = (\Phi_{\text{Bool}}, \Sigma_{\text{Bool}}, R_{\text{Bool}}, S_{\text{Bool}})$, where

$$\begin{aligned}
\Phi_{\text{Bool}} &= \{S_{\text{Bool}}\}, \\
\Sigma_{\text{Bool}} &= \{\wedge, \vee, \neg, x, y\}, \quad \text{and} \\
R_{\text{Bool}} &= \{S_{\text{Bool}} \to \wedge(S_{\text{Bool}}, S_{\text{Bool}}), S_{\text{Bool}} \to \vee(S_{\text{Bool}}, S_{\text{Bool}}), \\
&\qquad S_{\text{Bool}} \to \neg(S_{\text{Bool}}), S_{\text{Bool}} \to x, S_{\text{Bool}} \to y\}.
\end{aligned}$$

The sampling strategy uses the probabilities 0.3, 0.3, 0.1, 0.15, and 0.15 for the rules in R_{Bool} until 3 $\wedge$ or $\vee$ symbols have been sampled. Then, the probabilities are adjusted to 0.0, 0.0, 0.2, 0.4, and 0.4. If a $\neg$ has been sampled, then the probability for the third rule is set to zero and instead distributed onto rules 4 and 5. This scheme ensures that the number of binary operators ($\wedge$ or $\vee$) is limited to 3 and that no double negation can occur.

Expressions The grammar is $\mathcal{G}_{\text{Exp}} = (\Phi_{\text{Exp}}, \Sigma_{\text{Exp}}, R_{\text{Exp}}, S_{\text{Exp}})$, where

$$\begin{aligned}
\Phi_{\text{Exp}} &= \{S_{\text{Exp}}\}, \\
\Sigma_{\text{Exp}} &= \{+, \cdot, /, \sin, \exp, x, 1, 2, 3\}, \quad \text{and} \\
R_{\text{Bool}} &= \{S_{\text{Exp}} \to +(S_{\text{Exp}}, S_{\text{Exp}}), S_{\text{Exp}} \to \cdot(S_{\text{Exp}}, S_{\text{Exp}}), S_{\text{Exp}} \to /(S_{\text{Exp}}, S_{\text{Exp}}), \\
&\qquad S_{\text{Exp}} \to \sin(S_{\text{Exp}}), S_{\text{Exp}} \to \exp(S_{\text{Exp}}), \\
&\qquad S_{\text{Exp}} \to x, S_{\text{Exp}} \to 1, S_{\text{Exp}} \to 2, S_{\text{Exp}} \to 3\}.
\end{aligned}$$

The sampling strategy consists of three different sampling processes. Whenever we write 'sample', here, we mean sampling with uniform probabilities. First, we sample a binary operator b from $\{+, \cdot, /, L\}$. If $b = L$, we sample a single literal $\hat{x}_1$ from $\{x, 1, 2, 3\}$. Otherwise, we sample two literals x and y from $\{x, 1, 2, 3\}$ and set $\hat{x}_1 = b(x, y)$. Second, we sample a unary operator u from $\{\sin, \exp, L\}$ and a literal x from $\{x, 1, 2, 3\}$. If $u = L$, we set $\hat{x}_2 = x$, otherwise $\hat{x}_2 = u(x)$. Third, we sample an operator u from $\{\sin, \exp, U, B\}$ and repeat the first strategy to obtain a tree $\hat{y}$. If $u = B$, we set $\hat{x}_3 = \hat{y}$. If $u = U$, we repeat the second strategy to obtain $\hat{x}_3$. Otherwise, we set $\hat{x}_3 = u(\hat{y})$.

SMILES The grammar is $\mathcal{G}_{\text{SMILES}} = (\Phi_{\text{SMILES}}, \Sigma_{\text{SMILES}}, R_{\text{SMILES}}, S_{\text{SMILES}})$ where

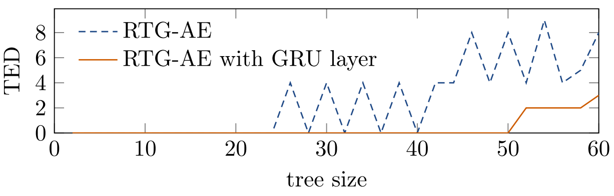

Fig. 4 The autoencoding error (tree edit distance) for vanilla RTG-AE (blue) and RTG-AE with a GRU layer for $g_2^{S_\oplus \to \oplus(B, S_\oplus)}$ (orange)

$$
\begin{aligned}
\Phi_{\text{SMILES}} &= \{S_{\text{SMILES}}, \mathcal{C}, A_1, A_2, E_1, E_2, \chi, H, Ch, O_1, O_2, B_1, B_2\}, \\
\Sigma_{\text{SMILES}} &= \{\text{smiles}, -, =, \equiv, /, \backslash, \#, \text{branch}, -_\circ, =_\circ, /_\circ, \backslash_\circ, -_b, =_b, \equiv_b, /_b, \backslash_b, \\
&\quad [], \text{hcount}, \text{charge}, @, @@, C, N, O, P, S, n, o, s, \\
&\quad C_{\text{Ali}}, N_{\text{Ali}}, O_{\text{Ali}}, S_{\text{Ali}}, P_{\text{Ali}}, F_{\text{Ali}}, Cl_{\text{Ali}}, Br_{\text{Ali}}, \\
&\quad c_{\text{Aro}}, n_{\text{Aro}}, o_{\text{Aro}}, s_{\text{Aro}}\}, \quad \text{and} \\
R_{\text{SMILES}} &= \{S_{\text{SMILES}} \to \text{smiles}(\mathcal{C}), \mathcal{C} \to -(\mathcal{C}, A_1), \mathcal{C} \to = (\mathcal{C}, A_1), \\
&\quad \mathcal{C} \to \equiv (\mathcal{C}, A_1), \mathcal{C} \to /(\mathcal{C}, A_1), \mathcal{C} \to \backslash(\mathcal{C}, A_1), \mathcal{C} \to \#(A_1), \\
&\quad A_1 \to \text{branch}(A_2, B_1^*, B_2^*), \\
&\quad A_1 \to \text{branch}(O_1, B_1^*, B_2^*), A_1 \to \text{branch}(O_2, B_1^*, B_2^*), \\
&\quad A_2 \to [](E_1, \chi?, H?, Ch?), A_2 \to [](E_2, \chi?, H?, Ch?), \\
&\quad E_1 \to C, E_1 \to N, E_1 \to O, E_1 \to P, E_1 \to S, \\
&\quad E_2 \to n, E_2 \to o, E_2 \to s, \\
&\quad \chi \to @, \chi \to @@, \\
&\quad H \to \text{hcount}, Ch \to \text{charge}, \\
&\quad O_1 \to C_{\text{Ali}}, O_1 \to N_{\text{Ali}}, O_1 \to O_{\text{Ali}}, O_1 \to S_{\text{Ali}}, \\
&\quad O_1 \to P_{\text{Ali}}, O_1 \to F_{\text{Ali}}, O_1 \to Cl_{\text{Ali}}, O_1 \to Br_{\text{Ali}}, \\
&\quad O_2 \to c_{\text{Aro}}, n_{\text{Aro}}, o_{\text{Aro}}, s_{\text{Aro}}, \\
&\quad B_1 \to -_\circ, B_1 \to =_\circ, B_1 \to /_\circ, B_1 \to \backslash_\circ, \\
&\quad B_2 \to -_b(\mathcal{C}), B_2 \to =_b (\mathcal{C}), B_2 \to \equiv_b (\mathcal{C}), \\
&\quad B_2 \to /_b(\mathcal{C}), B_2 \to \backslash_b(\mathcal{C})\}.
\end{aligned}
$$

In this grammar, the nonterminal $\mathcal{C}$ generates a chain of atoms. A_1 generates an atom which can branch off multiple chains, A_2 generates a single atom, E_1, and E_2 generate different kinds of element symbols, χ generates chirality, H generates a hydrogen count, Ch generates a charge, O_1 and O_2 generate element symbols in their role as aliphatic or aromatic atoms, B_1 generates a bond in the function of a ringbond, and B_2 generates a bond in the function of a branch. The logic of this grammar follows the SMILES specification (Weininger, 1988) in the version used by Kusner et al. (2017).

Pysort The pysort grammar is too unwieldy to be printed here in full. Instead, we point the interested reader to our source code repository https://gitlab.com/bpaassen/rtgae/-/blob/master/pysort_data.py or to the official Python documentation https://docs.python.org/3/library/ast.html.

Appendix D: List encoding

In Sect. 3.2, we noted that a GRU layer is preferable for decoding lists. To investigate this issue in more detail, we provide an extra experiment with the following grammar $\mathcal{G}_\oplus = (\Phi_\oplus, \Sigma_\oplus, R_\oplus, S_\oplus)$, which represents lists of binary numbers.

$$\begin{aligned}
\Phi_\oplus &= \{S_\oplus, B\},\\
\Sigma_\oplus &= \{\oplus, \#, 0, 1\},\\
R_\oplus &= \{S_\oplus \to \oplus(B, S_\oplus), S_\oplus \to \#(B), B \to 0, B \to 1\},
\end{aligned}$$

where $\oplus$ expresses list concatenation.

We now generate trees of the form $\#(0)$, $\oplus(1, \#(0))$, $\oplus(1, \oplus(1, \#(0)))$, and so on up to size 60. We train two RTG-AE models to auto-encode these trees, one vanilla model and one where we use a GRU layer for the decoding function $g_2^{S_\oplus \to \oplus(B,S_\oplus)}$. As hyperparameters, we use $n = 100$, $n_{\text{VAE}} = 8$, and learning rate 10^{-3}, same as for the Boolean and Expressions dataset.

Figure 4 displays the auto-encoding error versus tree length. We observe that the GRU model consistently achieves a lower error, especially for larger trees. Further, the RTG-AE model still gets caught in an endless loop for some cases: it produces an endless tree of the form $\oplus(1, \oplus(1, \oplus(1, \ldots)))$, whereas the GRU model perfectly auto-encodes all training trees.

This experiment illustrates the reason for our recommendation. In a list encoding, the danger of endless loops is very high because the same decoding gets repeated many times. In such a case, a feedforward layer for $g_2^{S_\oplus \to \oplus(B,S_\oplus)}$ has trouble to keep track how many remaining nodes still have to be decoded—even in this very simple example. Maintaining this memory over a lot of repetitions is simpler for a GRU layer.

Author contributions BP developed and implemented the proposed method, performed the experiments, acquired funding and wrote the initial draft. IK and KY supervised BP during development and experimental evaluation, and performed extensive revisions of the text.

Funding Open Access funding enabled and organized by Projekt DEAL. This work was supported by the German Research Foundation (DFG) under Grant Numbers PA 3460/1-1 and PA 3460/2-1.

Data availability All data and material required to reproduce the experiments is available at https://gitlab.com/bpaassen/rtgae/.

Code availability All code required to reproduce the experiments is available at https://gitlab.com/bpaassen/rtgae/.

Declarations

Conflict of interest The authors declare that they have no conflict of interest (beyond the funding acknowledged above).

Ethics approval Not applicable.

Consent to participate Not applicable.

Consent for publication Not applicable.

References

Aiolli, F., Da San, G. M., & Sperduti, A. (2015). An efficient topological distance-based tree kernel. *IEEE Transactions on Neural Networks and Learning Systems, 26*(5), 1115–1120. https://doi.org/10.1109/TNNLS.2014.2329331.

Allamanis, M., Chanthirasegaran, P., Kohli, P., & Sutton, C. (2017). Learning continuous semantic representations of symbolic expressions. In *Proceedings of the ICML* (pp. 80–88). http://proceedings.mlr.press/v70/allamanis17a.html.

Alon, U., Zilberstein, M., Levy, O., & Yahav, E. (2019). Code2vec: Learning distributed representations of code. In *Proceedings of the ACM programming languages* (Vol. 3). https://doi.org/10.1145/3290353.

Bacciu, D., Micheli, A., & Podda, M. (2019). Graph generation by sequential edge prediction. In M. Verleysen (Ed.), *Proceedings of the ESANN* (pp. 95–100). https://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2019-107.pdf.

Bille, P. (2005). A survey on tree edit distance and related problems. *Theoretical Computer Science, 337*(1), 217–239. https://doi.org/10.1016/j.tcs.2004.12.030.

Brainerd, W. S. (1969). Tree generating regular systems. *Information and Control, 14*(2), 217–231. https://doi.org/10.1016/S0019-9958(69)90065-5.

Burda, Y., Grosse, R. B., & Salakhutdinov, R. (2016). Importance weighted autoencoders. In *Proceedings of the ICLR*. arXiv:1509.00519.

Chi, Z. (1999). Statistical properties of probabilistic context-free grammars. *Computational Linguistics, 25*(1), 131–160. https://doi.org/10.5555/973215.973219.

Cho, K., van Merrienboer, B., Gülçehre, Ç ., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. In A. Moschitti (Ed.), *Proceedings of the EMNLP* (pp. 1724–1734). arXiv:1406.1078.

Collins, M., & Duffy, N. (2002). New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In P. Isabelle, E. Charniak, & D. Lin (Eds.), *Proceedings of the ACL* (pp.263–270). http://www.aclweb.org/anthology/P02-1034.pdf

Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., & Tommasi, M. (2008). Tree automata techniques and applications. Lyon, France: HAL open science. https://hal.inria.fr/ hal-03367725.

Dai, H., Tian, Y., Dai, B., Skiena, S., & Song, L. (2018). Syntax-directed variational autoencoder for structured data. In Y. Bengio, & Y. LeCun (Eds.), *Proceedings of the ICLR*. Retrieved from https://openreview.net/forum?id=SyqShMZRb.

Dyer, C., Kuncoro, A., Ballesteros, M., & Smith, N. A. (2016). Recurrent neural network grammars. In *Proceedings of the NAACL* (pp. 199–209). https://www.aclweb.org/anthology/N16-1024.pdf.

Gallicchio, C., & Micheli, A. (2013). Tree echo state networks. *Neurocomputing, 101,* 319–337. https://doi.org/10.1016/j.neucom.2012.08.017.

Hammer, B. (2002). Recurrent networks for structured data-A unifying approach and its properties. *Cognitive Systems Research, 3*(2), 145–165. https://doi.org/10.1016/S1389-0417(01)00056-0.

Jin, W., Barzilay, R., & Jaakkola, T. (2018). Junction tree variational autoencoder for molecular graph generation. In J. Dy & A. Krause (Eds.), *Proceedings of the ICML* (pp. 2323–2332). Retrieved from http://proceedings.mlr.press/v80/jin18a.html.

Kanade, A., Maniatis, P., Balakrishnan, G., & Shi, K. (2020). Learning and evaluating contextual embedding of source code. In H.D. III & A. Singh (Eds.), *Proceedings of the ICML* (pp. 5110–5121). Retrieved from http://proceedings.mlr.press/v119/kanade20a.html.

Kim, Y., Dyer, C., & Rush, A. M. (2019). Compound probabilistic contextfree grammars for grammar induction. In *Proceedings of the ACL* (pp. 2369–2385). Retrieved from https://www.aclweb.org/anthology/P19-1228.pdf.

Kingma, D., & Ba, J. (2015). Adam: A method for stochastic optimization. In Y. Bengio & Y. LeCun (Eds.), *Proceedings of the ICLR*. arXiv:1412.6980.

Kingma, D. P., & Welling, M. (2019). An introduction to variational autoencoders. *Foundations and Trends in Machine Learning, 12*(4), 307–392. https://doi.org/10.1561/2200000056.

Kipf, T., & Welling, M. (2017). Semi-supervised classification with graph convolutional networks. Y. Bengio & Y. LeCun (Eds.), *Proceedings of the ICLR*. Retrieved from https://openreview.net/forum?id=SJU4ayYgl.

Kusner, M. J., Paige, B., & Hernández-Lobato, J. M. (2017). Grammar variational autoencoder. D. Precup & Y. W. Teh (Eds.), *Proceedings of the ICML* (pp. 1945–1954). http://proceedings.mlr.press/v70/kusner17a.html.

Li, B., Cheng, J., Liu, Y., & Keller, F. (2019). Dependency grammar induction with a neural variational transition-based parser. In *Proceedings of the AAAI* (pp. 6658–6665). https://doi.org/10.1609/aaai.v33i01.33016658.

Liu, Q., Allamanis, M., Brockschmidt, M., & Gaunt, A. (2018). Constrained graph variational autoencoders for molecule design. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, & R. Garnett (Eds.), *Proceedings of the NeurIPS* (pp. 7795–7804). Retrieved from http://papers.nips.cc/paper/8005-constrained-graph-variational-autoencoders-for-molecule-design.

Loshchilov, I., & Hutter, F. (2016). CMA-ES for hyperparameter optimization of deep neural networks. arXiv, 1604.07269 . arXiv:1604.07269.

Micheli, A. (2009). Neural network for graphs: A contextual constructive approach. *IEEE Transactions on Neural Networks, 20*(3), 498–511. https://doi.org/10.1109/TNN.2008.2010350.

Paaßen, B., Grattarola, D., Zambon, D., Alippi, C., & Hammer, B. (2021). Graph edit networks. In S. Mohamed, K. Hofmann, A. Oh, N. Murray, & I. Titov (Eds.), *Proceedings of the ICLR*. https://openreview.net/forum?id=dlEJsyHGeaL.

Paaßen, B., Hammer, B., Price, T., Barnes, T., Gross, S., & Pinkwart, N. (2018). The continuous hint factory–providing hints in vast and sparsely populated edit distance spaces. *Journal of Educational Data Mining, 10*(1), 1–35.

Paaßen, B., Koprinska, I., & Yacef, K. (2020). Tree echo state autoencoders with grammars. In A. Roy (Ed.), *Proceedings of the IJCNN* (pp. 1–8). Retrieved from https://doi.org/10.1109/IJCNN48605.2020.9207165

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, & R. Garnett (Eds.), *Proceedings of the NeurIPS* (pp. 8026–8037). http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.

Pollack, J. B. (1990). Recursive distributed representations. *Artificial Intelligence, 46*(1), 77–105. https://doi.org/10.1016/0004-3702(90)90005-K.

Sanchez-Lengeling, B., & Aspuru-Guzik, A. (2018). Inverse molecular design using machine learning: Generative models for matter engineering. *Science, 361*(6400), 360–365. https://doi.org/10.1126/science.aat2663.

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., & Monfardini, G. (2009). The graph neural network model. *IEEE Transactions on Neural Networks, 20*(1), 61–80. https://doi.org/10.1109/TNN.2008.2005605.

Sperduti, A. (1994). Labelling recursive auto-associative memory. *Connection Science, 6*(4), 429–459. https://doi.org/10.1080/09540099408915733.

Sperduti, A., & Starita, A. (1997). Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks, 8*(3), 714–735. https://doi.org/10.1109/72.572108.

Sønderby, C. K., Raiko, T., Maaløe, L., Sønderby, S. K., Winther, O. (2016). Ladder variational autoencoders. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, & R. Garnett (Eds.), *Proceedings of the NeurIPS*. https://proceedings.neurips.cc/paper/2016/file/6ae07dcb33ec3b7c814df797cbda0f87-Paper.pdf.

Tai, K. S., Socher, R., & Manning, C. D. (2015). Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the ACL* (pp. 1556–1566).

Tiňo, P., & Hammer, B. (2003). Architectural bias in recurrent neural networks: Fractal analysis. *Neural Computation, 15*(8), 1931–1957. https://doi.org/10.1162/08997660360675099.

Turán, G. (1983). On the complexity of graph grammars. *Acta Cybernetica, 6,* 271–281.

Weininger, D. (1988). Smiles, a chemical language and information system. 1. Introduction to methodology and encoding rules. *Journal of Chemical Information and Computer Sciences, 28*(1), 31–36. https://doi.org/10.1021/ci00057a005.

Xu, K., Hu, W., Leskovec, J., & Jegelka, S. (2019). How powerful are graph neural networks? In T. Sainath, A. Rush, S. Levine, K. Livescu, & S. Mohamed (Eds.), *Proceedings of the ICLR*. arXiv:1810.00826.

Yogatama, D., Blunsom, P., Dyer, C., Grefenstette, E., & Ling, W. (2017). Learning to compose words into sentences with reinforcement learning. In Y. Bengio & Y. LeCun (Eds.), *Proceedings of the ICLR*. https://openreview.net/forum?id=Skvgqgqxe.

You, J., Ying, R., Ren, X., Hamilton, W., & Leskovec, J. (2018). GraphRNN: Generating realistic graphs with deep auto-regressive models. In J. Dy & A. Krause (Eds.), *Proceedings of the ICML* (pp. 5708–5717). Retrieved from http://proceedings.mlr.press/v80/you18a.html.

Zaremba, W., Kurach, K., & Fergus, R. (2014). Learning to discover efficient mathematical identities. In *Proceedings of the neurips* (Vol. 27, pp. 1278–1286). https://proceedings.neurips.cc/paper/2014/file/08419be897405321542838d77f855226-Paper.pdf.

Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X. (2019). A novel neural source code representation based on abstract syntax tree. In *Proceedings of the ICSE* (pp. 783-794). https://doi.org/10.1109/ICSE.2019.00086.

Zhang, K., & Shasha, D. (1989). Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing, 18*(6), 1245–1262. https://doi.org/10.1137/0218082.

Zhang, M., Jiang, S., Cui, Z., Garnett, R., Chen, Y. (2019). D-VAE: A variational autoencoder for directed acyclic graphs. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, & R. Garnett (Eds.), *Proceedings of the NeurIPS* (pp. 1586–1598). http://papers.nips.cc/paper/8437-d-vae-a-variational-autoencoder-for-directed-acyclic-graphs.

Zhao, S., Song, J., Ermon, S. (2019, Jul.). Infovae: Balancing learning and inference in variational autoencoders. In *Proceedings of the AAAI* (pp. 5885–5892). https://doi.org/10.1609/aaai.v33i01.33015885.