Tree Echo State Autoencoders with Grammars

Benjamin Paassen
School of Computer Science
The University of Sydney
Sydney, Australia
benjamin.paassen@sydney.edu.au

Irena Koprinska
School of Computer Science
The University of Sydney
Sydney, Australia
irena.koprinska@sydney.edu.au

Kalina Yacef
School of Computer Science
The University of Sydney
Sydney, Australia
kalina.yacef@sydney.edu.au

***Abstract*—Tree data occurs in many forms, such as computer programs, chemical molecules, or natural language. Unfortunately, the non-vectorial and discrete nature of trees makes it challenging to construct functions with tree-formed output, complicating tasks such as optimization or time series prediction. Autoencoders address this challenge by mapping trees to a vectorial latent space, where tasks are easier to solve, and then mapping the solution back to a tree structure. However, existing autoencoding approaches for tree data fail to take the specific grammatical structure of tree domains into account and rely on deep learning, thus requiring large training datasets and long training times. In this paper, we propose tree echo state autoencoders (TES-AE), which are guided by a tree grammar and can be trained within seconds by virtue of reservoir computing. In our evaluation on three datasets, we demonstrate that our proposed approach is not only much faster than a state-of-the-art deep learning autoencoding approach (D-VAE) but also has less autoencoding error if little data and time is given.**

***Index Terms*—echo state networks, regular tree grammars, reservoir computing, autoencoders, trees**

I. Introduction

Trees constitute an important data structure in a wide range of fields, describing diverse data such as computer programs [1], chemical molecules [2], or natural language [3]. In recent years, machine learning on these kinds of data has made considerable process, especially for classification and regression tasks [4]–[6]. In these cases, a machine learning model maps from trees to a scalar or vectorial output (*encoding*). The converse direction, mapping a vector back to a tree (*decoding*), however, is less well investigated, although such decoders would be highly useful for tasks such as generative models for trees, the optimization of tree structures, or time series prediction on trees [7]. In particular, a decoder for trees could help to optimize molecular structures [8], or to provide hints to students in intelligent tutoring systems [9].

Prior work on decoders for structured data can be roughly partitioned into two groups. First, decoders for full or acyclic graphs [10]–[13], which use deep recurrent neural networks to generate a graph one node or edge at a time until a full graph is completed. The drawback of these approaches is that they fail to take the specific structure of trees into account and thus may generate structures that are not trees. Furthermore, they do not take grammatical knowledge about the domain into account, which would be available for all aforementioned examples [1]–[3], and could thus be a useful prior.

The second group are decoders that take grammar information into account [8], [14], but are at present limited to string data instead of trees. Furthermore, both groups rely on deep neural networks for training which require large datasets and long training times.

Our key contribution in this paper are tree echo state autoencoders (TES-AE), a novel autoencoder architecture specifically dedicated to tree data, which uses grammatical knowledge and can be trained within seconds using a standard support vector machine solver [15], [16]. Our approach is based on tree echo state networks [4] for encoding and analogous networks for decoding, where we keep all neural network parameters fixed except for the final decoding layer. In our proposed model, this final layer decides which grammar rule to apply in each step of the decoding process. In our experiments on three datasets we show that our autoencoding approach can outperform deep variational autoencoders for acyclic graphs (D-VAE) [13] in terms of training time and autoencoding error, if little data and little training time is available. Further, we show that TES-AEs outperform sequential echo state networks for this application and that the TES-AE coding space is suitable for tree optimization, achieving similar results as [8].

In the following, we cover related work in more detail and recap background knowledge regarding regular tree grammars, before we describe our proposed architecture in depth, explain our experiments and results, and conclude with a summary of our findings.

II. Related Work

A. Tree Encoding

Most prior work on machine learning for trees can be grouped into neural network approaches (e.g. [4], [17], [18]) and tree kernel approaches (e.g. [19], [20]). In both cases, a tree $\hat{x}$ is first mapped to a vectorial representation $\phi(\hat{x}) = \vec{x}$, which is then used to complete a machine learning task, such as classification [17], regression [4], or dimensionality reduction [18]. We call the mapping ϕ an *encoder* for trees and we call $\vec{x}$ the *code* of $\hat{x}$ (refer to Figure 1, left). In more detail, recursive neural networks [4], [17], [18] encode trees by defining a function f which maps a node label and a (perhaps ordered) set of child encodings to an encoding for the parent node. The overall encoding ϕ is then computed via recursion.

Funding by the German Research Foundation (DFG) under grant number PA 3460/1-1 is gratefully acknowledged. Online supplement with source code at https://gitlab.com/bpaassen/tree_echo_state_autoencoders.

978-1-7281-6926-2/20/$31.00 ©2020 IEEE

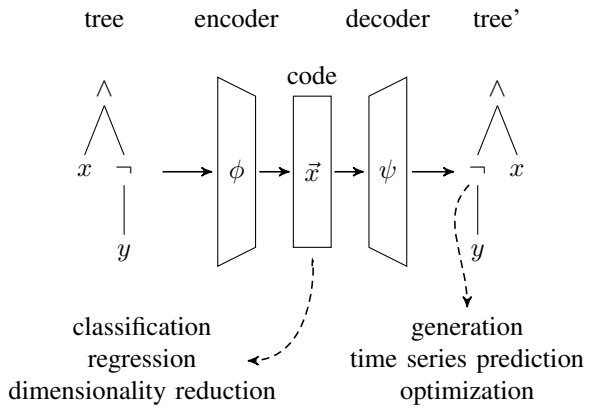

Fig. 1. An illustration of a tree encoder/decoder architecture and the applications for the code (classification, regression, dimensionality reduction) and the decoded tree (generation, optimization, time series prediction).

For example, the tree $\hat{x} = \wedge(x, \neg(y))$ from Figure 1 would be recursively encoded as $\phi(\hat{x}) = f(\wedge, \{\phi(x), \phi(\neg(y))\})$, where $\phi(x) = f(x, \emptyset)$ and $\phi(\neg(y)) = f(\neg, \{\phi(y)\}) = f(\neg, \{f(y, \emptyset)\})$. We follow this recursive encoding scheme in our work but adapt it slightly to be better aligned with a grammar.

B. Tree Decoding

While an encoder is sufficient to perform machine learning tasks with vectorial output, many interesting tasks require a *decoder* ψ as well, i.e. a mapping from the vector space back to the space of trees (refer to Figure 1, right). For example, we can address time series prediction by encoding a tree $\hat{x}$ as a vector $\phi(\hat{x})$, predicting the next state of the vector $\phi(\hat{x}) + \vec{\delta}$, and then decoding back to the next state of the tree $\psi(\phi(\hat{x}) + \vec{\delta})$ [9]; we can construct new trees by sampling a vector $\vec{x}$ in the latent space and then mapping back to a tree structure $\psi(\vec{x})$ [12]; and we can optimize trees by varying the representation in the latent space $\vec{x}$ such that some objective function $\ell(\psi(\vec{x}))$ on the decoded tree $\psi(\vec{x})$ is optimized [8].

Training a decoder for trees is considerably harder compared to an encoder because the dimensionality of the vector space (and hence the number of neurons in the model) needs to scale exponentially with the tree depth to distinguish all possible trees in a domain [21]. Accordingly, only few scholars to date have attempted to tackle the problem of tree decoding [7]. Most who did are concerned with the more general problem of graph decoding by generating a graph one node/edge at a time via a deep recurrent neural network [10]–[13]. In more detail, these approaches treat a graph as a sequence of node and edge insertions and attempt to reproduce this sequence with a recurrent neural network. The most applicable of these works to our setting are variational autoencoders for directed acyclic graphs (D-VAE) [13] because trees are a subclass of acyclic graphs and thus the architectural bias towards acyclicity should help D-VAEs in reconstructing trees.

Note that our proposed model is similar to these approaches in that we also equate a tree with a sequence of actions, namely a sequence of production rules in a regular tree grammar. However, we do not apply a recurrent neural network but follow the recursive structure of the tree. Further, by considering grammar rules instead of general node and edge insertions, our output trees are guaranteed to be syntactically correct whereas existing graph decoders may violate syntactic rules or produce data that is not tree-formed at all.

With respect to the reliance on grammars, our approach resembles the work of [8] who also suggested to guide a decoder by a grammar. Also like [8], we train our networks to achieve autoencoding, i.e. we wish to train a ψ that acts as an inverse of an encoder ϕ on the training data. However, we consider tree data instead of string data and use recursive networks instead of (time-)convolutional networks.

C. Echo State Networks

A final and crucial difference to all previous work lies in our choice of training scheme. While all aforementioned approaches use gradient descent across the entire network, we base our approach on the reservoir computing literature (e.g. [22], [23]). More precisely, we use a slightly varied version of the tree echo state network [4] as encoder and decoder, where all internal parameters are initialized randomly, then pre-processed to ensure eventual forgetting of inputs [22], but kept fixed afterwards. We only train the final layer that decides which grammar rule to take in each step of the decoding. Because of this, our training problem becomes convex and easy to solve. In particular, we can use a straightforward support vector machine solver [15], [16] to train the output layer. Our main contribution to the reservoir computing literature is that we propose not only an encoder, but a decoder model for trees.

D. Regular tree grammars

Our approach strongly relies on regular tree grammars [24], [25], such that we now take some time to describe them in more detail, albeit in a simplified notation to ease understanding.

First, we define a *tree* $\hat{x}$ over some finite alphabet Σ as an expression $x(\hat{y}_1, \ldots, \hat{y}_k)$, where $x \in \Sigma$ and where $\hat{y}_1, \ldots, \hat{y}_k$ are also trees over Σ, which we call the *children* of $\hat{x}$. Note that k may be zero, in which case we call the tree a *leaf*. For example, for $\Sigma = \{\wedge, \vee, \neg, x, y\}$, $x()$, $\vee(x(), y())$, $\neg(x())$, and $\wedge()$ are all trees over Σ, where $x()$ and $\wedge()$ are leaves. Per convention, we omit the empty brackets for leaves.

Note that our definition of trees is very liberal and includes many instances that may be nonsensical according to the rules of the domain. To restrict the space of possible trees to a more sensible subset, we use *regular tree grammars*. We define a regular tree grammar as a 4-tuple $\mathcal{G} = (\Phi, \Sigma, R, S)$, where Φ is a finite set of nonterminal symbols, Σ is a finite set of terminal symbols, $S \in \Phi$ is a special nonterminal symbol which we call the *starting* symbol, and R is a finite set of production rules of the form $A \to x(B_1, \ldots, B_k)$ where $A, B_1, \ldots, B_k \in \Phi$ and $x \in \Sigma$.

We say that a tree $\hat{y}$ over $\Phi \cup \Sigma$ can be *derived in one step* via grammar $\mathcal{G}$ from another tree $\hat{x}$ over $\Phi \cup \Sigma$, if there exists a production rule $A \to x(B_1, \ldots, B_k)$ and a leaf A in $\hat{x}$, such that replacing A with $x(B_1, \ldots, B_k)$ yields $\hat{y}$. Generalizing this definition, we say that a tree $\hat{y}$ can be derived in T steps via grammar $\mathcal{G}$ from another tree $\hat{x}$, if there exists a sequence of trees $\hat{z}_0 \to \ldots \to \hat{z}_T$ such that $\hat{z}_0 = \hat{x}$, $\hat{z}_T = \hat{y}$, and $\hat{z}_t$ can be derived in one step via grammar $\mathcal{G}$ from $\hat{z}_{t-1}$ for all $t > 0$. Finally, we define the *tree language* $\mathcal{L}(\mathcal{G})$ as the set of all trees $\hat{x}$ over Σ which can be derived in T steps from the starting symbol S for any $T \in \mathbb{N}$. As an example, consider the regular tree grammar in Figure 3, left. The tree $\wedge(x, \neg(y))$ can be derived in 4 steps from S via the sequence $S \to \wedge(S, S) \to \wedge(x, S) \to \wedge(x, \neg(S)) \to \wedge(x, \neg(y))$.

An important property of regular tree grammars is that they can be parsed efficiently using tree automata [24], [25]. This is especially easy to see for a subclass of regular tree grammars, which we call *deterministic*. We define a regular tree grammar as deterministic if no two production rules have the same right-hand-side. For these grammars, we can parse a tree $\hat{x} = x(\hat{y}_1, \ldots, y_k)$ via the following recursive function: First, we parse all children of $\hat{x}$. This will return a nonterminal symbol B_i for every child $\hat{y}_i$ and a sequence of rules deriving $\hat{y}_i$ from B_i. After that, we simply have to check whether a rule of the form $A \to x(B_1, \ldots, B_k)$ exists in our grammar. If so, we return the nonterminal symbol A and the concatenation of this rule and all rule sequences for the children. If not, the parse ends because the tree is not part of the tree language. We utilize this scheme later for encoding in Algorithm 1.

III. Method

Our aim in this paper is to construct an autoencoder for trees that exploits grammatical knowledge for the tree domain. More precisely, for a given regular tree grammar $\mathcal{G}$ we would like to obtain an encoder $\phi : \mathcal{L}(\mathcal{G}) \to \mathbb{R}^n$ for some $n \in \mathbb{N}$ and a decoder $\psi : \mathbb{R}^n \to \mathcal{L}(\mathcal{G})$, such that for as many trees $\hat{x} \in \mathcal{L}(\mathcal{G})$ as possible, $\hat{x}$ is close to $\psi(\phi(\hat{x}))$. To achieve this goal, we introduce two approaches. We start with a sequence-to-sequence learning approach following the architecture of [26] and then continue with an approach based on tree echo state networks [4], which we describe in terms of encoding, decoding, and training.

A. Sequence-to-sequence learning

Sequence-to-sequence learning is a neural network architecture introduced by [26], which translates an input sequence to an output sequence, potentially of different length. The architecture features two recurrent neural networks, an encoding network $f : \mathbb{R}^l \times \mathbb{R}^n \to \mathbb{R}^n$, a decoding network $g : \mathbb{R}^l \times \mathbb{R}^n \to \mathbb{R}^n$, and an output function $h : \mathbb{R}^n \to \mathbb{R}^l$ for some input dimensionality l and encoding dimensionality n. The encoding network translates the input time series $\vec{y}_1, \ldots, \vec{y}_T \in \mathbb{R}^l$ into an encoding vector $\vec{x}_T$ by means of the equation $\vec{x}_t = f(\vec{y}_t, \vec{x}_{t-1})$ where $\vec{x}_0 = \vec{0}$, i.e. a vector of n zeros. This encoding is then used to generate the output time series $\vec{z}_1, \ldots, \vec{z}_{T'}$ as follows. We first set the initial decoding state as $\tilde{x}_1 = \vec{x}_T$ and then generate the first output as $\vec{z}_1 = h(\tilde{x}_1)$. All remaining decoding states are generated via $\tilde{x}_t = g(\vec{z}_{t-1}, \tilde{x}_{t-1})$ and all remaining outputs via $\vec{z}_t = h(\tilde{x}_t)$ until $\vec{z}_t$ is a special end-of-sequence token, whereupon the process stops.

To apply the sequence-to-sequence learning framework to tree data, we first translate an input tree $\hat{x}$ into a sequence of production rules, which we represent via one-hot codes, then encode this sequence via the encoder network, decode it to a sequence of one-hot codes, translate these back to production rules, and finally produce the decoded tree using these rules.

Figure 2 illustrates the approach for the example tree $\wedge(x, \neg(y))$ and the example grammar from Figure 3. Recall that our example tree can be derived from the starting symbol S via the sequence $S \to^{\mathbf{1}} \wedge(S, S) \to^{\mathbf{4}} \wedge(x, S) \to^{\mathbf{3}} \wedge(x, \neg(S)) \to^{\mathbf{5}} \wedge(x, \neg(y))$, where we indexed each arrow by its corresponding production rule according to the numbering from Figure 3. Accordingly, the tree is equivalent to the production rule sequence $(\mathbf{1}, \mathbf{4}, \mathbf{3}, \mathbf{5})$, which we represent by one-hot codes in the second row of Figure 2. We then apply the encoding network f four times to achieve an overall encoding $\vec{x}_4$ of our input sequence, which we then plug in our decoder as initial state $\tilde{x}_1$. From this initial state, our output function h predicts the first element $\vec{y}_1$ of our output rule sequence, which is then fed back into the decoding network g to generate the second state $\tilde{x}_2$, and so on until h predicts the special end-of-sequence token $(0, 0, 0, 0, 0, 1)$.

In our case, we implement both f and g as recurrent neural networks with the equations $\vec{x}_t = f(\vec{y}_t, \vec{x}_{t-1}) = \tanh\left(\boldsymbol{U} \cdot \vec{y}_t + \boldsymbol{W} \cdot \vec{x}_{t-1}\right)$ and $\tilde{x}_t = g(\vec{z}_{t-1}, \tilde{x}_{t-1}) = \tanh\left(\boldsymbol{U} \cdot \vec{z}_{t-1} + \boldsymbol{W} \cdot \tilde{x}_{t-1}\right)$, and the output function as a linear function $\vec{z}_t = h(\tilde{x}_t) = \boldsymbol{V} \cdot \tilde{x}_t$. Note that the matrices $\boldsymbol{U}$, $\boldsymbol{W}$, and $\boldsymbol{V}$ are parameters of our model. Following the reservoir computing paradigm [22], we do not train the matrices $\boldsymbol{U}$ or $\boldsymbol{W}$ but initialize them as cycle reservoir with jumps [23] and then keep them fixed. Note that we use the same matrices $\boldsymbol{U}$ and $\boldsymbol{W}$ for f and g. Next, we generate for each tree in the training data the decoding state sequence $\tilde{x}_1, \ldots, \tilde{x}_{T+1}$ via teacher forcing, i.e. $\tilde{x}_t = \tanh\left(\boldsymbol{U} \cdot \vec{y}_{t-1} + \boldsymbol{W} \cdot \tilde{x}_{t-1}\right)$, using $\vec{y}_{t-1}$ as input argument instead of $\vec{z}_{t-1}$. Finally, we train the matrix $\boldsymbol{V}$ via linear regression on the training data $\left\{(\tilde{x}_t, \vec{y}_t) | t \in \{1, \ldots, T\}\right\}$.

While this approach is already functional in principle, we expect it to fail for reasonably large input trees. This is because our network needs to remember rule applications a long time ago to correctly predict the next production rule. Echo state networks, however, focus on intense short-term memory instead of long-term memory [22], [27]. Accordingly, we now attempt to reduce the number of time steps between encoding and decoding by working along the tree structure instead of flattening it to a sequence beforehand.

B. Tree Encoding

To encode a tree, we follow the parsing scheme for (deterministic) regular tree grammars outlined in the background section. More formally, let $\mathcal{G} = (\Phi, \Sigma, R, S)$ be a regular tree grammar. Then, for each grammar rule $r = \big(A \to$

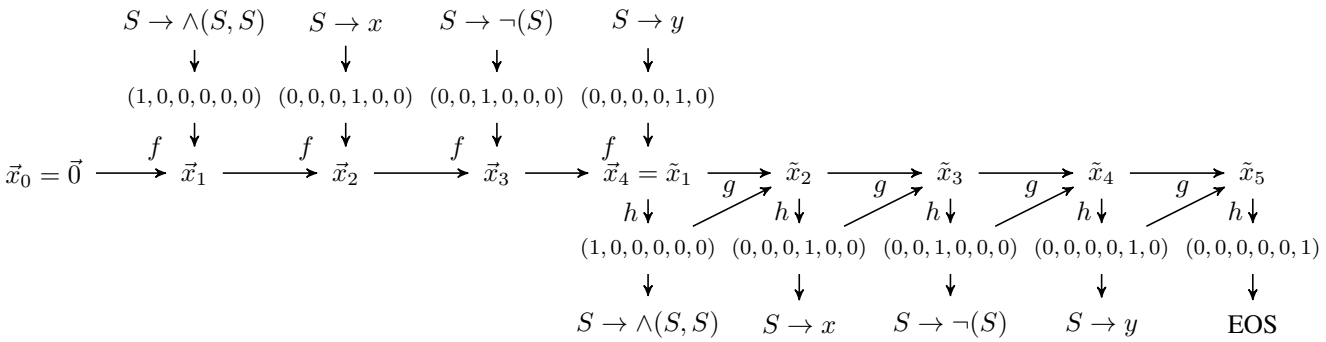

Fig. 2. An illustration of the sequence-to-sequence autoencoding architecture for the example tree $\wedge(x, \neg(y))$ and the regular tree grammar from Figure 3. Top: The rule sequence generating the tree; second row: the translation of the rule sequence into one-hot-coding; third row: the sequence of encoding and decoding states; last row: the output series of one-hot codings.

$x(B_1, \ldots, B_k)\big) \in R$, we define a function $f_r : \mathbb{R}^{n \times k} \to \mathbb{R}^n$, such that we can construct the encoding $\phi(\hat{x})$ of a tree $\hat{x} = x(\hat{y}_1, \ldots, \hat{y}_k)$ recursively as

$$\phi(\hat{x}) = f_r\Big(\phi(\hat{y}_1), \ldots, \phi(\hat{y}_k)\Big) \tag{1}$$

The precise algorithm for encoding is outlined in Algorithm 1. An example is shown in Figure 3. In the example, we start with the entire tree $\wedge(x, \neg(y))$ and pass downward through the tree until we reach the first leaf, which is x. We parse this leaf using the fourth grammar rule $S \to x$, such that our encoding function returns the the nonterminal S, the rule sequence $(\mathbf{4})$, and the vector encoding $\phi(x) = f_4()$. We perform the same scheme for the leaf y, yielding the nonterminal S, the rule sequence $(\mathbf{5})$, and the encoding $f_5()$. We then proceed with the partially parsed subtree $\neg(S)$, which we can parse using the third rule $S \to \neg(S)$, yielding the nonterminal S, the rule sequence $(\mathbf{3}, \mathbf{5})$, and the encoding $f_3(f_5())$. This leaves the tree $\wedge(S, S)$, which we can parse using the first rule, yielding the nonterminal S, the rule sequence $(\mathbf{1}, \mathbf{4}, \mathbf{3}, \mathbf{5})$, and the overall encoding $\phi(\wedge(x, \neg(y))) = f_1(f_4(), f_3(f_5()))$.

Algorithm 1 An algorithm to encode and parse trees according to a deterministic regular tree grammar $\mathcal{G} = (\Phi, \Sigma, R, S)$ and encoding functions f_r for each rule $r \in R$. 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.

function ENCODE(a tree $\hat{x} = x(\hat{y}_1, \ldots, \hat{y}_k)$)
  for $j \in \{1, \ldots, k\}$ **do**
    $B_j, (r_{j,1}, \ldots, r_{j,T_j}), \vec{y}_j \leftarrow$ ENCODE($\hat{y}_j$).
  end for
  if $\exists A \in \Phi : A \to x(B_1, \ldots, B_k) \in R$ **then**
    $r \leftarrow \big(A \to x(B_1, \ldots, B_k)\big)$.
    return $A, (r, r_{1,1}, \ldots, r_{k,T_k}), f_r(\vec{y}_1, \ldots, \vec{y}_k)$.
  else
    Error; $\hat{x}$ is not in $\mathcal{L}(\mathcal{G})$.
  end if
end function

We implement each of the functions f_r as a single-layer feedforward neural network, i.e.

$$f_r(\vec{y}_1, \ldots, \vec{y}_k) = \tanh\Big(\boldsymbol{W}_1^r \cdot \vec{y}_1 + \ldots + \boldsymbol{W}_k^r \cdot \vec{y}_k + \vec{b}^r\Big) \tag{2}$$

where the $n \times n$ matrices $\boldsymbol{W}_1^r, \ldots, \boldsymbol{W}_k^r$ and the bias vector $\vec{b}^r \in \mathbb{R}^n$ are parameters of f_r. Following the reservoir computing paradigm, we do not train these parameters but keep them fixed [22]. In more detail, we initialize a $\beta \in (0, 1]$ fraction of the entries for each matrix as standard normally distributed random numbers, and then enforce a spectral radius of $\rho \in (0, 1)$. We fill the bias vectors with normally distributed random numbers with zero mean and standard deviation ρ. Note that the coding dimensionality n, as well as the sparsity β and the spectral radius ρ are hyper-parameters of our approach.

We remark in passing that the reservoir computing paradigm would suggest that each of the reservoir matrices $\boldsymbol{W}_j^r$ is universal [22], [23]. Accordingly, one could assume that it suffices to initialize *one* reservoir matrix and re-use it across the entire model instead of initializing a separate matrix for each argument of each rule. However, using the same reservoir for all input arguments collapses Equation 2 to $\tanh\big(\boldsymbol{W} \cdot (\vec{x}_1 + \ldots + \vec{x}_k)\big)$, which is now an order-invariant function with respect to the input and, as such, strictly less powerful. Still, we will consider this version as a baseline in our experiments later on.

C. Tree Decoding

For decoding, we emulate the production process of a regular tree grammar. We begin with the starting symbol S and the vectorial code $\vec{x}$ for the tree to be decoded. Then, we let a classifier $h_S : \mathbb{R}^n \to R$ decide which of the possible rules $r = S \to x(B_1, \ldots, B_k)$ with S on the left-hand-side we should apply. Next, we decode $\vec{x}$ into vectorial codes $\vec{y}_1, \ldots, \vec{y}_k$ for the children. For this step, we use decoding functions $g_j^r : \mathbb{R}^n \to \mathbb{R}^n$ that should extract the information for the jth child from $\vec{x}$. We then repeat this scheme recursively until all nonterminal symbols are decoded. We present the decoding scheme more formally in Algorithm 2.

As an example, consider Figure 4. We start at the top with the vector code for the entire tree and the starting nonterminal

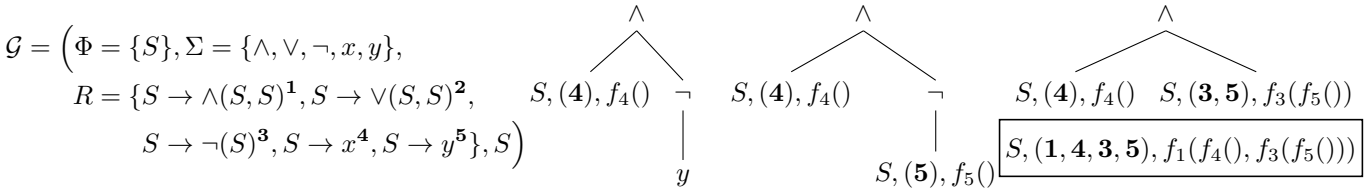

Fig. 3. An illustration of the encoding algorithm 1 for the tree $\wedge(x, \neg(y))$. Left: The tree grammar with enumerated rules (number labels in upper index). From center to right: Each step of the encoding process with the final result highlighted with a box. During encoding, each node is replaced with a triple of a nonterminal label, a sequence of grammar rules (here as numbers), and a vectorial encoding (here abstracted via function symbols).

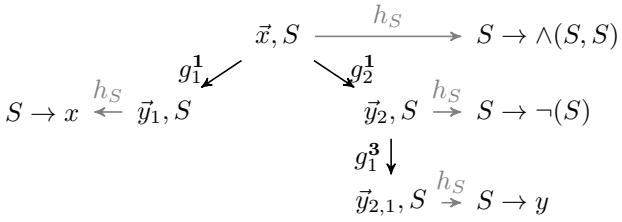

Fig. 4. An illustration of the decoding algorithm 2 for the tree $\wedge(x, \neg(y))$ (from top to bottom).

S. The classifier h_S then selects the first rule $S \to \wedge(S,S)$ (top right) to apply. Based on this selection, we know that we need to use the decoding functions $g_1^{\mathbf{1}}$ and $g_2^{\mathbf{1}}$ to obtain vectorial codings $\vec{y}_1$ and $\vec{y}_2$ for the new children. We then apply the same scheme to the newly created vector codes and nonterminals until the entire tree is decoded.

Algorithm 2 An algorithm to decode vectors to trees according to a regular tree grammar $\mathcal{G} = (\Phi, \Sigma, R, S)$, classifiers $h_A : \mathbb{R}^n \to R$ for each nonterminal $A \in \Phi$, and decoding functions g_j^r for each rule $r \in R$ and each of its arguments j. The function receives a vector and a nonterminal symbol as input and returns a decoded tree.

function DECODE(a vector $\vec{x} \in \mathbb{R}^n$, a nonterminal $A \in \Phi$)
  $r = \big(A \to x(B_1, \ldots, B_k)\big) \leftarrow h_A(\vec{x})$.
  for $j \in \{1, \ldots, k\}$ **do**
    $\vec{y}_j \leftarrow g_j^r(\vec{x})$.
    $\hat{y}_j \leftarrow$ DECODE($\vec{y}_j$, B_j).
    $\vec{x} \leftarrow \vec{x} - \vec{y}_j$.
  end for
  return $x(\hat{y}_1, \ldots, \hat{y}_k)$.
end function

Just as before, we implement the decoding functions $g_j^r : \mathbb{R}^n \to \mathbb{R}^n$ using single-layer feedforward neural networks, i.e.: $g_j^r(\vec{x}) = \tanh(\boldsymbol{W}_j^r \cdot \vec{x} + \vec{b}_j^r)$, where the matrices $\boldsymbol{W}_j^r$ and the bias vectors $\vec{b}_j^r$ are parameters of the model. We apply the same initialization scheme for the matrices $\boldsymbol{W}_j^r$ and the vectors $\vec{b}_j^r$ as during encoding, and keep the parameters fixed after initialization.

D. Training

In our model, we only need to train the rule classifiers h_A for every nonterminal A. For training these classifiers, we need to know the encoding vectors $\vec{x}$ for every nonterminal during the decoding process. Fortunately, we can compute these vectors for our training data using teacher forcing. In particular, recall that Algorithm 1 does not only yield the encoding for the tree, but also a rule sequence that generates the tree. This sequence contains the desired outputs for all our classifiers. Furthermore, we can use this sequence to decide which rules to apply during decoding, such that we can complete the entire decoding process without relying on the classifiers' outputs. We describe the details of this computation in Algorithm 3. Note that this algorithm executes Algorithm 1 first and then executes a modified version of Algorithm 2 where the decision of the rule classifiers h_A is replaced by the ground truth rule sequence. The training data sets $\mathcal{D}_A$ can be accumulated across an entire training set of trees and then be used to train the rule classifiers h_A. In the example from Figure 4, the training data would be $\mathcal{D}_S = \{(\vec{x}, \mathbf{1}), (\vec{y}_1, \mathbf{4}), (\vec{y}_2, \mathbf{3}), (\vec{y}_{2,1}, \mathbf{5})\}$ because we should execute the first rule when we encounter the encoding $\vec{x}$, the fourth rule when we encounter $\vec{y}_1 = g_1^{\mathbf{1}}(\vec{x})$, the third rule when we encounter $\vec{y}_1 = g_2^{\mathbf{1}}(\vec{x} - \vec{y}_1)$, and the fifth rule when we encounter $\vec{y}_{2,1} = g_1^{\mathbf{3}}(\vec{y}_2)$.

As classifiers h_A for each nonterminal $A \in \Phi$ we employ a standard support vector machine [15].

IV. Experiments

In our experimental evaluation we compare four models. First, a variational autoencoder for directed acyclic graphs (D-VAE) as proposed by [13]; second, the sequence-to-sequence autoencoder from Section III-A, which we call echo-state autoencoder (*ES-AE*); third, our tree echo state auto-enocoder with shared reservoir matrix across all rules (*S-TES-AE*); and fourth, our tree echo state autoencoder with separate weight matrices for each rule (*TES-AE*). Generally, we expect the D-VAE model to do better than all our reservoir computing models because it can adjust all weights instead of just the output weights. However, we expect that training a D-VAE takes much longer. Between our echo state models, we expect the TES-AE to do better than the S-TES-AE and the S-TES-AE to do better than the ES-AE, in alignment with our arguments in Section III.

Algorithm 3 An algorithm to generate training data for the rule classifiers h_A from a tree $\hat{x}$ according to a regular tree grammar $\mathcal{G} = (\Phi, \Sigma, R, S)$. The algorithm receives a tree $\hat{x}$ as input and returns a set of training data for each nonterminal symbol $A \in \Phi$.

function TRAIN(a tree $\hat{x}$)
  $A, (r_1, \ldots, r_T), \vec{x} \leftarrow$ ENCODE($\hat{x}$).
  Initialize a stack $\mathcal{S}$ with $\vec{x}$ on top.
  Initialize an empty set $\mathcal{D}_A$ for each $A \in \Phi$.
  for $t \leftarrow 1, \ldots, T$ **do**
    Let $r_t = A \to x(B_1, \ldots, B_k)$.
    Pop $\vec{x}_t$ from the top of $\mathcal{S}$.
    Add $(\vec{x}_t, r_t)$ to $\mathcal{D}_A$.
    for $j \leftarrow k, \ldots, 1$ **do**
      $\vec{y}_j \leftarrow g_j^{r_t}(\vec{x}_t)$.
      Push $\vec{y}_j$ onto $\mathcal{S}$.
      $\vec{x}_t \leftarrow \vec{x}_t - \vec{y}_j$.
    end for
  end for
  return $\{\mathcal{D}_A | A \in \Phi\}$.
end function

TABLE I
STATISTICS OF THE THREE DATASETS.

statistic	Boolean	expressions	pysort
no. of trees	500	500	51
no. of nonterminals $\lvert\Phi\rvert$	1	1	12
no. of terminals $\lvert\Sigma\rvert$	5	9	54
no. of rules $\lvert R\rvert$	5	9	54
avg. tree size	5.3	9.06	64.41

We evaluate each model on three datasets. First, a dataset of Boolean expressions (*Boolean*), which we generate by applying random rules of the grammar in Figure 3 until at most three binary operators (and/or) are present.

Second, a dataset of function expressions (*expressions*) as described by [8]. The grammar for this dataset is $\mathcal{G} = (\{S\}, \{+, *, /, \sin, \exp, x, 1, 2, 3\}, \{S \to +(S,S), S \to *(S,S), S \to /(S,S), S \to \sin(S), S \to \exp(S), S \to x, S \to 1, S \to 2, S \to 3\}, S)$. We sample expressions by adding one binary operator to one unary operator to one unary with a binary argument, e.g. $3 * x + \sin(x) + \exp(2/x)$, which is consistent with the training data generated by [8].

Third, a dataset of 51 python programs (*pysort*) implementing the insertion sort algorithm or parts of it. The dataset can be found in the online supplement[1]. The grammar is the full python language grammar as documented on https://docs.python.org/3/library/ast.html. The statistics for all datasets are listed in Table I.

For the D-VAE model, we used the authors' reference implementation[2]. We implemented all echo state models in python using *scikit-learn* [16] as support vector machine solver. All implementations are available in the online supplement[1]. We ran all experiments on a consumer-grade laptop with Intel core i7 CPU.

[1]https://gitlab.com/bpaassen/tree_echo_state_autoencoders
[2]https://github.com/muhanzhang/D-VAE

A. Autoencoding

We first evaluate the models in terms of their capacity for autoencoding. As measure of performance, we consider the root mean square error (RMSE), in particular the formula $\sqrt{\frac{1}{m}\sum_{i=1}^{m} d(\hat{x}_i, \psi[\phi(\hat{x}_i)])^2}$, where $\hat{x}_i$ are the test trees, ϕ and ψ are the en- and decoding functions of the respective model, and d is the tree edit distance [28].

For the D-VAE model we used the same experimental parameters as in the original paper [13] because the long training times made hyperparameter optimization prohibitive. However, we used less epochs (50) and higher learning rate (10^{-3}) to further limit training time. For the echo state models, we fixed the number of neurons to 256 to achieve a fair comparison between the models and optimized all other hyperparameters on extra validation data. In particular, for *Boolean* and *expressions* we sampled 100 additional training trees and 100 additional test trees specifically for hyperparameter optimization. For *pysort* we randomly removed 5 training trees and 5 test trees from the main dataset for hyperparameter optimization. The optimization itself was a random search with 50 trials for *Boolean* and *expressions* and 20 trials for *pysort*. The precise ranges for each hyper-parameter can be found in the online supplement[1].

For the evaluation itself, we performed a cross-validation with 20 folds on *Boolean* and *expressions* and 10 folds on *pysort*. To keep training times manageable, we evaluated the D-VAE model only once with a 10% test data split.

We report the RMSEs for all models and all datasets in Table II. As expected, the S-TES-AE model clearly outperforms the ES-AE model on all data sets and the TES-AE model outperforms the S-TES-AE model on the first two datasets. These differences are statistically significant in a Wilcoxon sign-rank test with $p < 0.05$ after Bonferroni correction. On the pysort dataset, the performance of TES-AE and S-TES-AE is statistically indistinguishable. Surprisingly, the D-VAE model performed worse than both tree echo state models on all datasets, which is likely caused by the small amount of training data, the short training time, and, most of all, the lack of grammatical knowledge encoded in the network. In particular, we observe that only 34% of the decoded Boolean formulae, 9% of the decoded mathematical expressions, and none of the decoded python programs conformed to the respective grammar. However, the architectural bias of D-VAE was sufficient to at least achieve a tree structure for 100% of the Boolean formulae, 95% for the mathematical expressions, and three of five python programs.

To check how training time influenced the results, we trained the D-VAE model on the Boolean dataset again with 300 epochs (just above 2.5 hours of training time), resulting in an RMSE of 3.70 and 42% grammatical correctness, which is still considerably worse than the TES-AE model.

TABLE II
ACCURACY OF ALL MODELS IN TERMS OF RMSE ON AUTOENCODING THE TEST DATA (± STANDARD DEVIATION, EXCEPT FOR D-VAE, WHICH WAS EVALUATED ONLY ONCE)

dataset	D-VAE	ES-AE	S-TES-AE	TES-AE
Boolean	4.62	3.64 ± 0.44	3.25 ± 0.39	2.84 ± 0.49
expressions	5.81	3.87 ± 0.61	2.65 ± 0.23	1.69 ± 0.21
pysort	52.07	64.86 ± 7.00	16.97 ± 4.30	17.49 ± 5.04

TABLE III
TRAINING TIME IN SECONDS (± STANDARD DEVIATION, EXCEPT FOR D-VAE WHICH IT WAS EVALUATED ONLY ONCE).

dataset	D-VAE	ES-AE	S-TES-AE	TES-AE
Boolean	757.1	1.26 ± 0.04	2.44 ± 0.07	3.29 ± 0.22
expressions	1201.76	0.85 ± 0.01	3.37 ± 0.10	5.06 ± 0.06
pysort	13991.5	0.47 ± 0.02	0.63 ± 0.11	10.83 ± 0.76

Regarding runtimes (refer to Table III), we observe that ES-AE and S-TES-AE are comparably fast on the Boolean and pysort datasets but the latter is factor 3 slower on the expressions dataset. Furthermore, TES-AE is considerably slower on all datasets than S-TES-AE (factors 1.5 on the first two datasets and factor 15 on the pysort dataset). This is to be expected as setting up more parameters including a matrix decomposition for the spectral radius computation for each parameter matrix is expensive. Further, the parameter matrices for cycle reservoir with jumps [23] are sparser than our Gaussian random number initialization, making ES-AE and S-TES-AE even faster. In all cases, however, the overall runtime remains within a few seconds time. This is in stark contrast to the D-VAE model, which took over 10 minutes to train on the Boolean dataset, over 20 minutes for the expressions dataset, and over 3 hours for the pysort dataset.

B. Optimization

Next, we evaluated the capacity for tree optimization in the coding space. For the Boolean dataset, we considered the logical evaluation of the formula, assuming that x is true and y is false. We assign a score of 0 if the formula evaluates to false and otherwise as the number of fulfilled $\wedge$ terms in the formula. For example, $\wedge(x, \neg(y))$ would evaluate to 1 because there is one fulfilled 'and' but $\wedge(y, \wedge(x, x))$ would evaluate to 0 because the entire formula evaluates to false.

For the expressions dataset, we used the performance measure of [8], i.e. we evaluated the arithmetic expressions for 1000 linearly spaced values of x between -10 and $+10$ and computed the logarithm of one plus the mean square error to the ground truth function $1/3 + x + \sin(x \cdot x)$.

Because our coding space was quite high-dimensional ($n = 256$), we did not perform Bayesian optimization as suggested by [8] but used a Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) instead, namely the reference implementation of the python `cma` package[3]. To be comparable with [8], we limited the computational budget to the same value, namely 750 overall function evaluations, which we distributed onto 15 iterations with 50 evaluations each.

TABLE IV
THE OPTIMIZED TREE AND ITS SCORE FOR ALL MODELS FOR THE BOOLEAN AND EXPRESSIONS DATASETS. FOR BOOLEAN, HIGHER SCORES ARE BETTER AND FOR EXPRESSIONS, LOWER SCORES ARE BETTER.

model	optimal expression	score
	Boolean	
ES-AE	$\wedge(\wedge(\vee(\wedge(x,x),x),x),\wedge(x,\wedge(x,\wedge(x,x))))$	6
s-TES-AE	$\wedge(\wedge(\vee(y,x),x),\wedge(x,x))$	3
TES-AE	$\vee(\neg(\vee(y,\wedge(\wedge(x,x),x))),\wedge(\wedge(\wedge(y,x)),x))$	3
	expressions	
ES-AE	+(x, /(1, *(1, 3)))	0.391
s-TES-AE	+(/(1, 3), +(x, sin(*(x, x))))	0
TES-AE	+(x, +(sin(3), sin(*(x, x))))	0.036

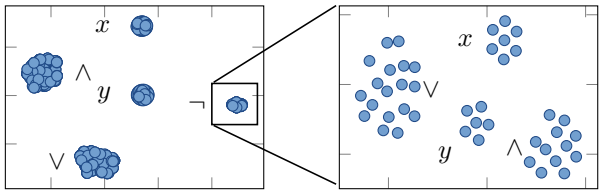

Fig. 5. Left: A t-SNE visualization of the coding space of the TES-AE model on the Boolean dataset. Each cluster in the visualization is labelled with the root symbol of all trees in the cluster. Right: A t-SNE visualization of only trees with ¬ at the root; clusters are labelled with the symbol below the root.

The results are shown in Table IV. Note that the results for D-VAE are missing because CMA-ES failed to generate any grammatical tree which could have been evaluated. Regarding the results of the echo state models, we note that the sequential echo state autoencoder (ES-AE) performed best on the Boolean dataset by extrapolating beyond the training data and using seven binary operators instead of the three that were present in the training data. The TES-AE model also extrapolated, but with less success. Only the S-TES-AE model remained within the boundaries of the training data and achieved the best possible value within it.

Regarding the expression dataset, both TES-AE variations found a solution at least as good as the grammar variational autoencoder of [8] and the s-TES-AE model even found the ground truth. Overall, the s-TES-AE model appears to be best suited for optimization on these tasks.

C. Coding Spaces

If we inspect the encoding spaces of the TES-AE model in more detail, we observe clusters dependent on root symbol of the tree. For example, Figure 5 (left) shows a t-SNE dimensionality reduction of the Boolean dataset as encoded by the TES-AE model with each cluster labelled with the root symbol. In Figure 5 (right), we observe that the ¬ cluster

[3]https://github.com/CMA-ES/pycma

further spreads into clusters according to the symbol just below the root. This fractal coding is consistent with prior work on recurrent networks with small weights, which have been shown to code fractally based on the most recent symbol [29].

V. Conclusion

In this paper, we introduced tree echo state autoencoders (TES-AE), a novel neural network architecture to implement autoencoding for trees without the need for deep learning. In particular, we used regular tree grammars to express our trees as sequences of grammar rules and then employed echo state networks and tree echo state networks for encoding and decoding. In our experiments on three datasets, we found that a TES-AE outperformed a variational auto-encoder for acyclic graphs (D-VAE) in terms of autoencoding error on small datasets with limited training time. Further, we showed that TES-AE significantly outperform a sequential version of the model (ES-AE) and that separate parameters for each grammar rule outperform shared parameters. Our results also showed that a few seconds sufficed to train our model even for a large grammar and large trees, whereas D-VAE training, even with a small number of epochs, took ten minutes to several hours. Finally, we observed that optimization in the TES-AE coding space performed similarly compared to past reference results [8].

Future research could investigate how well our autoencoders are suitable to time series prediction, how memory capacity results translate to the tree domain, how to apply our architecture to trees with real-valued nodes, and whether our proposed echo state sequence-to-sequence learning model using echo state networks is suitable to solve sequence tasks that currently require deep learning.

References

[1] A. Aho, M. Lam, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Boston, MA, USA: Addison Wesley, 2006.

[2] D. Weininger, "Smiles, a chemical language and information system. 1. introduction to methodology and encoding rules," *Journal of Chemical Information and Computer Sciences*, vol. 28, no. 1, pp. 31–36, 1988.

[3] K. Knight and J. Graehl, "An overview of probabilistic tree transducers for natural language processing," in *Computational Linguistics and Intelligent Text Processing*, A. Gelbukh, Ed. Springer Berlin Heidelberg, 2005, pp. 1–24.

[4] C. Gallicchio and A. Micheli, "Tree echo state networks," *Neurocomputing*, vol. 101, pp. 319 – 337, 2013.

[5] T. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proc. ICLR*, 2017. [Online]. Available: https://openreview.net/forum?id=SJU4ayYgl

[6] B. Paaßen, C. Gallicchio, A. Micheli, and B. Hammer, "Tree edit distance learning via adaptive symbol embeddings," in *Proc. ICML*, vol. 80, 2018, pp. 3973–3982. [Online]. Available: http://proceedings.mlr.press/v80/paassen18a.html

[7] B. Paaßen, C. Gallicchio, A. Micheli, and A. Sperduti, "Embeddings and representation learning for structured data," in *Proc. ESANN*, 2019, pp. 85–94. [Online]. Available: https://arxiv.org/abs/1905.06147

[8] M. J. Kusner, B. Paige, and J. M. Hernández-Lobato, "Grammar variational autoencoder," in *Proc. ICML*, 2017, pp. 1945–1954. [Online]. Available: http://proceedings.mlr.press/v70/kusner17a.html

[9] B. Paaßen, B. Hammer, T. Price, T. Barnes, S. Gross, and N. Pinkwart, "The continuous hint factory - providing hints in vast and sparsely populated edit distance spaces," *Journal of Educational Datamining*, vol. 10, no. 1, pp. 1–35, 2018. [Online]. Available: https://jedm.educationaldatamining.org/index.php/JEDM/article/view/158

[10] Q. Liu, M. Allamanis, M. Brockschmidt, and A. Gaunt, "Constrained graph variational autoencoders for molecule design," in *Proc. NeurIPS*, 2018, pp. 7795–7804. [Online]. Available: http://papers.nips.cc/paper/8005-constrained-graph-variational-autoencoders-for-molecule-design.pdf

[11] J. You, R. Ying, X. Ren, W. Hamilton, and J. Leskovec, "GraphRNN: Generating realistic graphs with deep auto-regressive models," in *Proc. ICML*, 2018, pp. 5708–5717. [Online]. Available: http://proceedings.mlr.press/v80/you18a.html

[12] D. Bacciu, A. Micheli, and M. Podda, "Graph generation by sequential edge prediction," in *Proc. ESANN*, 2019. [Online]. Available: https://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2019-107.pdf

[13] M. Zhang, S. Jiang, Z. Cui, R. Garnett, and Y. Chen, "D-vae: A variational autoencoder for directed acyclic graphs," in *Proc. NeurIPS*, 2019, pp. 1586–1598. [Online]. Available: http://papers.nips.cc/paper/8437-d-vae-a-variational-autoencoder-for-directed-acyclic-graphs.pdf

[14] H. Dai, Y. Tian, B. Dai, S. Skiena, and L. Song, "Syntax-directed variational autoencoder for structured data," in *Proc. ICLR*, 2018. [Online]. Available: https://openreview.net/forum?id=SyqShMZRb

[15] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, Sep 1995.

[16] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[17] A. Sperduti and A. Starita, "Supervised neural networks for the classification of structures," *IEEE Transactions on Neural Networks*, vol. 8, no. 3, pp. 714–735, 1997.

[18] B. Hammer, A. Micheli, A. Sperduti, and M. Strickert, "Recursive self-organizing network models," *Neural Networks*, vol. 17, no. 8, pp. 1061 – 1085, 2004.

[19] M. Collins and N. Duffy, "New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron," in *Proc. ACL*, 2002, pp. 263–270. [Online]. Available: http://www.aclweb.org/anthology/P02-1034.pdf

[20] F. Aiolli, G. Da San Martino, and A. Sperduti, "An efficient topological distance-based tree kernel," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 26, no. 5, pp. 1115–1120, 2015.

[21] B. Hammer, "Recurrent networks for structured data - A unifying approach and its properties," *Cognitive Systems Research*, vol. 3, no. 2, pp. 145 – 165, 2002.

[22] H. Jaeger and H. Haas, "Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication," *Science*, vol. 304, no. 5667, pp. 78–80, 2004.

[23] A. Rodan and P. Tiňo, "Simple deterministically constructed cycle reservoirs with regular jumps," *Neural Computation*, vol. 24, no. 7, pp. 1822–1852, 2012.

[24] W. S. Brainerd, "Tree generating regular systems," *Information and Control*, vol. 14, no. 2, pp. 217 – 231, 1969.

[25] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi, *Tree Automata Techniques and Applications*. inria gforge, 2008. [Online]. Available: http://tata.gforge.inria.fr/

[26] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. NIPS*, 2014, pp. 3104–3112. [Online]. Available: http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural

[27] I. Farkaš, R. Bosák, and P. Gergeľ, "Computational analysis of memory capacity in echo state networks," *Neural Networks*, vol. 83, pp. 109 – 120, 2016.

[28] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM Journal on Computing*, vol. 18, no. 6, pp. 1245–1262, 1989.

[29] P. Tiňo and B. Hammer, "Architectural bias in recurrent neural networks: Fractal analysis," *Neural Computation*, vol. 15, no. 8, pp. 1931–1957, 2003.